The MagPi ESSENTIALS

MAKE GAMES WITH PYTHON

CREATE YOUR OWN ENTERTAINMENT WITH *Raspberry Pi*

Written by **Sean M. Tracey**

The MagPi

Number one for Raspberry Pi

raspberrypi.org/magpi

SUBSCRIBE TODAY FROM **£13 / $37.50**

THE OFFICIAL RASPBERRY PI MAGAZINE

NEW! YOUR OFFICIAL RASPBERRY PI MAGAZINE
100 pages of hacking & making
STREAM LIVE FROM YOUR PI
MAKE A HIGH-SCORE TABLE IN SCRATCH
BUILD A FRIDGE MONITOR
Also inside:
COLLISIONS IN PYGAME
SONIC PI V2.6 REVIEWED
HOW TO INSPIRE CHILDREN WITH
BUILD A COMPUTER VISION SEC
THE ONLY MAGAZINE WRITTEN BY
NEW! YOUR OFFICIAL RASPBERRY PI MAGAZINE
100 pages of hacking & making
BUILD A
BUILD A WEB-POWERED PLANT WATERER
MAKE MUSIC WITH THE PIANO HAT
THE SENSE HAT
THE OFFICIAL PI TOUCH SCREEN
Also inside:
LEARN CLASSES IN PYTHON
MANCHESTER MAKEFEST REPO
PI WINS BIG AT FESTIVAL OF CO
MORE OF YOUR AMAZING PROJE
THE ONLY MAGAZINE WRITTEN BY
YOUR OFFICIAL RASPBERRY PI MAGAZINE
100 pages of hacking & making
TIME-LAPSE PHOTOGRAPHY
ASTRO PI UPDATE
PIXEL ART WITH THE SENSE HAT
Also inside:
MAKE GAMES WITH RASPBERRY
MASTER PIMORONI'S SKYWRITE
CREATE AMAZING MUSIC WITH S
REVIEWED: SCRATCH, RASPBIA
THE ONLY MAGAZINE WRITTEN BY
FREE RASPBERRY PI WITH THIS ISSUE
The MagPi
The official Raspberry Pi magazine
Issue 40 Christmas 2015
raspberrypi.org/magpi
#PIZERO
Raspberry Pi Zero is a real $5 computer. Learn to code while you play Minecraft (or do just about anything else you can dream up)
RASPBERRY PI ZERO
FREE W
100 PAGES OF HACKING & MAKING
PLUS: ASTRO PI IS ENROUTE TO THE IN

raspberrypi.org/magpi

WELCOME TO MAKE GAMES WITH PYTHON

While countless millions of us take great pleasure spending hours racking up high scores in our favourite games, few of us are ever exposed to the delights of making them in the first place. It's far from easy, but learning to code your own shoot-'em-up is infinitely more satisfying than beating any end-of-level boss.

Although this book is designed to help you learn many of the essential skills you'll need to make games with Python and Pygame on your Raspberry Pi, it's by no means definitive. Frankly, you could read a dozen books on the subject and still not have the skills you need to succeed. As with most things, nothing replaces good old-fashioned practice. I should know: I have 30 cookery books lining my shelf and I still burnt my toast this morning.

Making games is a brilliant way to learn to code, though, so I hope this book helps you to get started on your next big adventure.

Russell Barnes
Managing Editor, Raspberry Pi

FIND US ONLINE raspberrypi.org/magpi **GET IN TOUCH** magpi@raspberrypi.org

The MagPi

Available on the App Store

Get it on Google play

EDITORIAL
Managing Editor: **Russell Barnes**
russell@raspberrypi.org
Technical Editor: **David Whale**
Sub Editors: **Lorna Lynch (with Laura Clay & Phil King)**

DESIGN
Critical Media: **criticalmedia.co.uk**
Head of Design: **Dougal Matthews**
Designers: **Lee Allen, Mike Kay**

DISTRIBUTION
Seymour Distribution Ltd
2 East Poultry Ave,
London
EC1A 9PT | **+44 (0)207 429 4000**

SUBSCRIPTIONS
Select Publisher Services Ltd
PO Box 6337
Bournemouth
BH1 9EH | **+44 (0)1202 586 848**

recycle

In print, this product is made using paper sourced from sustainable forests and the printer operates an environmental management system which has been assessed as conforming to ISO 14001.

The MagPi magazine is published by Raspberry Pi (Trading) Ltd., Mount Pleasant House, Cambridge, CB3 0RN. The publisher, editor and contributors accept no responsibility in respect of any omissions or errors relating to goods, products or services referred to or advertised in the magazine. Except where otherwise noted, content in this magazine is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0). ISSN: 2051-9982.

The MagPi ESSENTIALS

CONTENTS

05 [CHAPTER ONE]
SHAPES & PATHS WITH PYGAME

18 [CHAPTER TWO]
ANIMATING SHAPES & PATHS

28 [CHAPTER THREE]
TAKING CONTROL OF THE KEYBOARD & MOUSE

42 [CHAPTER FOUR]
YOUR FIRST GAME

56 [CHAPTER FIVE]
PYGAME SOUNDBOARD

68 [CHAPTER SIX]
PHYSICS & FORCES

80 [CHAPTER SEVEN]
PHYSICS & COLLISIONS

94 [CHAPTER EIGHT]
BUILDING CLASSES

114 [CHAPTER NINE]
THE ALIENS ARE TRYING TO KILL ME!

130 [CHAPTER TEN]
THE ALIENS ARE HERE & THEY'RE COMING IN WAVES!

[SEAN M. TRACEY]

Sean calls himself a technologist, which is a fancy way of saying he still hasn't decided what he wants to do with technology – other than everything. Sean has spent his career trying to avoid getting 'proper' jobs, and as such has had a hand in making a variety of fun and interesting projects, including a singing statue of Lionel Richie, wearable drum kits, chopstick bagpipes, time-telling hats, and a life-sized Elvis Presley robot, to name only a few.
sean.mtracey.org

The MagPi ESSENTIALS

[CHAPTER ONE]
SHAPES & PATHS WITH PYGAME

We are going to learn how to make a game on our Raspberry Pi from the ground up. In the first chapter, we learn the basics.

In this book, we are going to learn to make games on the Raspberry Pi with Pygame. We'll look at drawing, animation, keyboard and mouse controls, sound, and physics. Each chapter will add to our knowledge of Raspberry Pi game development, allowing us both to understand the games we play, and to create almost anything our imaginations can come up with.

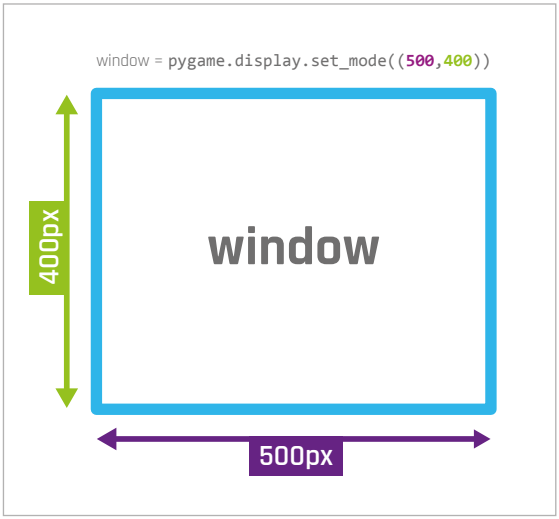
This book isn't for absolute programming beginners, but it's not far from it: we're going to assume that you've written some simple Python (or similar) programs in the past, and are able to do things like creating files and get around your Pi's filesystem without too much difficulty. If you haven't set up your Pi and are a little lost on how to go about it, there are lots of easy-to-follow guides on the web which will help bring you up to speed. You could point your web browser to **raspberrypi.org/resources** to get started.

In the first chapter, we're going to look at drawing and colouring various shapes in a window. This isn't quite *Grand Theft Auto V*, admittedly, but drawing shapes is the first step in building just about anything.

To start off, open your preferred text editor, create a new file, insert the following code into it and save it as **hello.py**: Let's run that code and see what it does. In your terminal window, enter **python hello.py**. If all has gone well, a new window will have opened showing you a

Download
magpi.cc/1jQhJYe

```
import pygame

pygame.init()

window = pygame.display.set_mode((500, 400))

while True:

    pygame.draw.rect(window, (255,0,0),
                             (0, 0, 50, 30))

    pygame.display.update()
```



Left Here we can see how each variable in `window` affects the application window's shape and size. Width always comes before height

red square on a black background in the top-left corner of the window. We've just created our first Pygame program: let's walk through it.

Understanding hello.py

The first two lines of our first program are very simple: all we've done is told Python that we want to use Pygame. **`import pygame`** loads all of the Pygame code into our script, so we don't have to write all of that code ourselves. Pygame is designed to make the creation of games and interactive software easy. **`pygame.init()`** tells Pygame that we're ready to start using it.

Let's look at the third line of code:

```
window = pygame.display.set_mode((500, 400))
```

`window` is the parameter we're going to use to tell our Pygame program about how it should look when it runs; each parameter affects the application window's shape and size. Note that here, width always comes before height. **`window`** is also the parameter that we'll use to tell other lines of code the surface on which they should draw shapes

and set colours. With `window`, we're calling the `set_mode` function of Pygame's display module: the latter is responsible for how the game window and surface (an informal term for the pixels we'll be manipulating) behaves. We're passing a tuple (which we can think of as a special list of things – in this case, a list of numbers) to `set_mode()` to tell it how big we want our game window to be. In this case, the application window is 500 pixels wide by 400 pixels tall. If we pass numbers that are bigger, the game window will be bigger; if we pass numbers that are smaller, the game window will be smaller.

The next few lines are where we make our program draw shapes on that window. When programs run, they execute their code, and when they're finished, they close themselves. That's fine unless, of course, you want your program to be interactive, or to draw or animate shapes over time, which is exactly what we need from a game. So, in order to keep our program from exiting, we make a `while` loop and put all our code inside. The `while` loop will never finish because `True` is always `True`, so we can keep running our program and drawing our shapes for as long as we like.

The first thing we do in our `while` loop is draw a rectangle. A rectangle is the simplest shape that we can draw in Pygame:

```
pygame.draw.rect(window, (255,0,0), (0,0,50,30))
```

The parameters at the end are telling Pygame where we want to draw our rectangle, the colour we want our rectangle to be, how we want to draw it, and how big we want it to be.

In our `hello.py` program, we've told Pygame to draw a rectangle in our window – or, at least, the surface we create with our `window` parameter. Next, we told Pygame what colour we wanted our rectangle to be by passing it through a tuple (a special list of numbers) representing how much red, green, and blue the final colour should have in it. We use red, green, and blue as these are the three colours your screen combines to create every shade you can see on it. 0 means that none of that colour should be used in the shape; 255 means that the maximum amount of colour should be in that shape. We told our rectangle that it should be the colour `(255, 0, 0)`, which is pure red.

[PYGAME]

Pygame is installed on Raspbian by default. Find documentation detailing all its features at **pygame.org/docs**

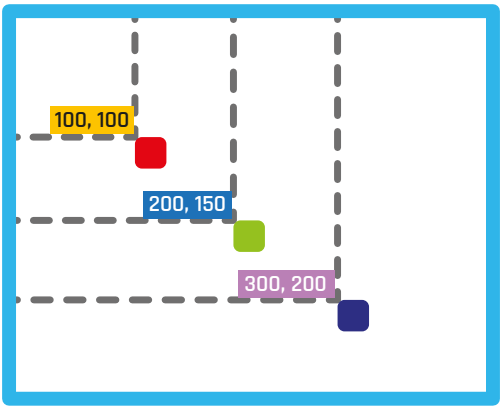
If we had told it to be `(255, 0, 255)`, it would have been a bright purple, because it's being drawn with the maximum amount of red and the maximum amount of blue. If we had told our rectangle to be coloured `(100, 100, 100)`, it would be a dark grey, because all of the colours would be equal.

After we've passed through a colour for our rectangle to be, we have to tell it where it should go and how big it should be. We do this by passing a tuple of four numbers. The first number is an X coordinate, which set out how far from the left side of the window the left edge of our rectangle should be. The second number is a Y coordinate; this tells the rectangle how far from the top of our window the top edge it should sit. The third number gives the width of our rectangle, and the fourth number defines its height. So, for example, if we wanted our rectangle to be 50 pixels from the left side of the window, 100 pixels from the top of our window, 20 pixels wide and 80 pixels tall, we would pass `(50, 100, 20, 80) to pygame.draw.rect()`.

Please note that the order never changes. If you tell Pygame how big you want the rectangle to be when it's expecting a colour or vice versa, the program may crash, so take your time.

Our last line in `hello.py` is nice and simple: it tells Pygame that we're done drawing shapes for the moment and that it can now refresh the window. This saves our Pi having to draw and redraw the screen for every shape that we've created; instead, it can get them all drawn in one go.

```
pygame.display.rect(window,(255,0,0),(100,100,50,50))
pygame.display.rect(window,(0,255,0),(200,150,50,50))
pygame.display.rect(window,(0,0,255),(300,200,50,50))
```



Below Here's a clear look at what each variable does to the shape we're drawing

Adding more shapes

We've successfully drawn one shape, so let's draw a few more. We'll draw some squares around the screen and mess around with their properties a little bit. There's no need to create a new file, so we'll stick with `hello.py` for now. Edit the `while` loop so it's the same as the following:

```
while True:

    pygame.draw.rect(window, (255,0,0),
                             (100, 100, 50, 50))
    pygame.draw.rect(window, (0,255,0),
                             (150, 100, 50, 50))
    pygame.draw.rect(window, (0,0,255),
                             (200, 100, 50, 50))

    pygame.display.update()
```

Now we should have three squares: red, blue, and green. So far, this is nice and simple, but those squares are placed right next to each other. What would happen if they were to overlap? Let's find out. Change your code once more to the following:

```
while True:

    pygame.draw.rect(window, (255,0,0),
                             (0, 0, 50, 50))
    pygame.draw.rect(window, (0,255,0),
                             (40, 0, 50, 50))
    pygame.draw.rect(window, (0,0,255),
                             (80, 0, 50, 50))

    pygame.display.update()
```

This time we get two rectangles and a square, but that is not what we asked for. So, what has gone wrong? When we execute our code, it works through what it has to draw, and where it has to put it, line-by-line. If one item is drawn and then another is drawn over it or on top of part of it, then we can no longer see what's beneath that second shape. The pixels of the shape drawn first

[LINE WIDTH]

When drawing a rectangle or ellipse, you have the choice of passing a line width. If you don't, the shape will be filled solid.

are lost when we overlap it with another shape. If we change the order of our code, we can see this effect in action. Cut and paste the code for the second square so that it becomes the third square drawn, like so:

```
while True:

    pygame.draw.rect(window, (255,0,0),
                             (0, 0, 50, 50))
    #pygame.draw.rect(window, (0,255,0),
                             #(40, 0, 50, 50))FROM HERE
    pygame.draw.rect(window, (0,0,255),
                             (80, 0, 50, 50))
    pygame.draw.rect(window, (0,255,0),
                             (40, 0, 50, 50)) #TO HERE

    pygame.display.update()
```

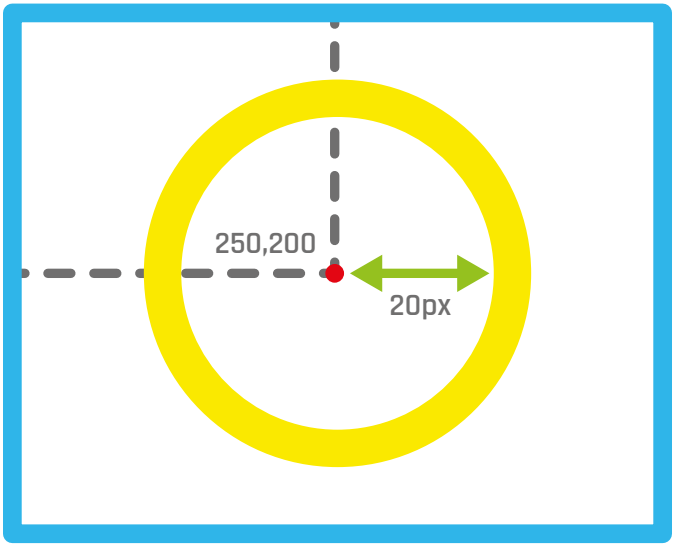
Now we the code apparently produces rectangle, square, rectangle. This is because the red and blue squares were drawn first and then the green square was drawn over the top of them. The red and blue squares re still there in their entirety, but we can't see all of them, so they look like rectangles.

Pygame allows us to do a great deal more than merely draw rectangles: we can make all kinds of other shapes too, including circles, ellipses, and paths (which are made up of many lines between multiple points).

Drawing circles

The process of drawing a circle is much like drawing a square except that, instead of passing a width and a height, we pass a radius and a point, around which we draw our circle. So, for example, if we wanted to draw a yellow circle in the middle of our window with a diameter of 40 pixels, we would use the following code to replace the code in the original `while` loop in `hello.py`:

```
pygame.draw.circle(window,(255,255,0),(250,200),20,1)
```



Left Here's how the variables enable you to draw a circle

```
while True:

        #Just like before to help us remember
        #pygame.draw.circle(WHERE TO DRAW, (RED, GREEN,
BLUE), (X COORDINATE, Y COORDINATE), RADIUS, HEIGHT,
WIDTH)

        pygame.draw.circle(window, (255,255,0),
                                   (250, 200), 20, 0)

        pygame.display.update()
```

Just like drawing a rectangle, we tell Pygame on which surface we want to draw our circle, what colour we want it to be, and where it should go. The radius is specific to drawing this particular shape. You might have noticed

that we put a `0` after our radius; this is a value used to determine the width of the line that draws our circle. If we pass `0`, the circle is filled; but if we pass `2`, for instance, we get a 2-pixel-wide line with an empty centre:

```
while True:

    #Filled
    pygame.draw.circle(window,(255,255,0),
                               (200, 200), 20, 0)

    #Not filled
    pygame.draw.circle(window,(255,255,0),
                               (300, 200), 20, 2)

    pygame.display.update()
```

What about ellipses? They are a slightly strange cross between drawing rectangles and circles. As we did when we drew a rectangle, we pass an X coordinate, a Y coordinate, a width, and a height, but we end up with an elliptical shape. Let's draw an ellipse or two.

```
while True:

    pygame.draw.ellipse(window, (255, 0, 0),
                                (100, 100, 100, 50))
    pygame.draw.ellipse(window, (0, 255, 0),
                                (100, 150, 80, 40))
    pygame.draw.ellipse(window, (0, 0, 255),
                                (100, 190, 60, 30))

    pygame.display.update()
```

Just as before, run your code. You should now see three ellipses: one red, one green, and one blue. Each should be a different size. If you wanted to

visualise how these shapes were generated, you could draw rectangles using the same coordinates as you used to draw an ellipse and it would fit perfectly inside that box. As you may have guessed, this means you can also make circles by using `pygame.draw.ellipse` if the width and height parameters are the same.

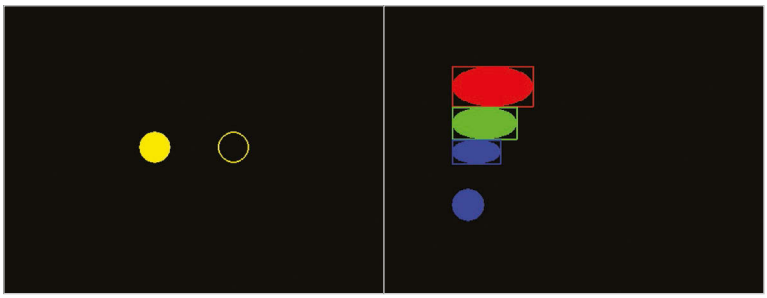
```
while True:

    pygame.draw.rect(window, (255, 0, 0),
                             (100, 100, 100, 50), 2)
    pygame.draw.ellipse(window, (255, 0, 0),
                                (100, 100, 100, 50))

    pygame.draw.rect(window, (0, 255, 0),
                             (100, 150, 80, 40), 2)
    pygame.draw.ellipse(window, (0, 255, 0),
                                (100, 150, 80, 40))

    pygame.draw.rect(window, (0, 0, 255),
                             (100, 190, 60, 30), 2)
    pygame.draw.ellipse(window, (0, 0, 255),
                                (100, 190, 60, 30))

    #Circle
    pygame.draw.ellipse(window, (0, 0, 255),
                                (100, 250, 40, 40))

    pygame.display.update()
```

A new path

We have covered rectangles, squares and circles, but what if we want to draw a triangle, a pentagon, a hexagon or an octagon? Unfortunately, there aren't functions for every kind of shape, but we can use paths. Paths allow us to draw irregular shapes by defining points in space, joining them up with lines, and filling in the space we've created. This is a little more complex, so it's time to move on from our original

[TUPLE]

A tuple is like a list, but unlike a standard list, a tuple's contents can't be changed (it's immutable). **python.org/docs**



Right When drawing a circle, the last variable lets us know if the circle should be filled in or not

Far Right Ellipses in the rectangles that bound them

`hello.py` program. Create a new file, call it `paths.py`, and save it with the following text inside:

```
import pygame

pygame.init()
window = pygame.display.set_mode((500, 400))

while True:
        pygame.display.update()
```

This is simply our bare-bones Pygame app again. If you want to make a copy of this for experimenting without breaking anything, now would be a good time to do so.

Every path is made of connected lines, but, before we start joining things up, let's draw a couple of standalone lines to familiarise ourselves with them. We can do this with `pygame.draw.line()`. Edit `paths.py` so your `while` loop reads as follows:

```
while True:

    pygame.draw.line(window, (255,255,255),
                             (0, 0), (500, 400), 1)

    pygame.display.update()
```

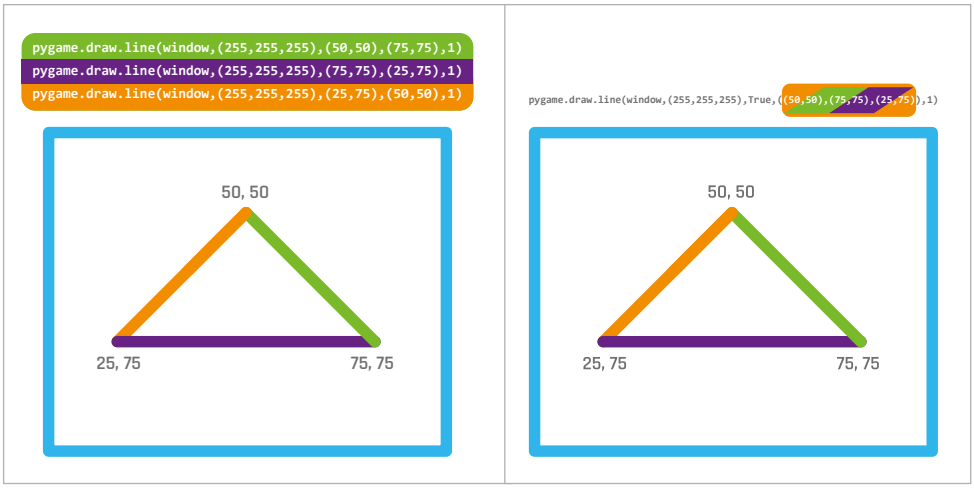
If you run this code now, you'll see a one-pixel-wide white line going from the top left to the bottom right of our Pygame window. The parameters we pass to `pygame.draw.line` start off the same way rectangles and ellipses do. We first tell Pygame where we want to draw the shape and then we choose a colour. Now, things change a little. The next argument is a tuple with the X and Y coordinates for where we want our line to start, and the third argument is a tuple with the X and Y coordinates for where we want our line to end. These are the two points between which our line will be drawn. The final argument is the width of the line is being drawn in pixels.

With this, we can now create shapes by defining points in our window. Let's draw that triangle we talked about earlier:

```
while True:
    pygame.draw.line(window, (255,255,255),
                             (50, 50), (75, 75), True)
    pygame.draw.line(window, (255,255,255),
                             (75, 75), (25, 75), True)
    pygame.draw.line(window, (255,255,255),
                             (25, 75), (50, 50), True)

    pygame.display.update()
```

You should have an image of a white triangle with a 1px edge. However, this code is rather lengthy: so many things, like the colour or the width of the line, are written multiple times. There is, however, a more concise way to achieve the result we want. All we need is `pygame.draw.lines()`.Whereas `pygame.draw.line()` lets us draw a line between two points, `pygame.draw.lines()` enables us to draw a sequence of lines between numerous points. Each XY-coordinate point will be joined up to the next XY-coordinate point, which will be joined up to the next XY-coordinate point, and so on.

After running the code on the next page, you'll see that the resulting triangle is exactly the same, except that we produced it from one line of code instead of three. You might have noticed that we didn't actually close the triangle: Pygame did it for us. Just before we pass the points



Above You can make a triangle from three separate lines

Above Right This triangle is made up of one line with multiple points. Follow the colours to see which variable is which

for our shape to be drawn from, we can pass either a `True` of a `False` value that will let Pygame know that we want it to close our shapes for us. Change it to `False` and we get the first two lines of our shape, but not the third. If we want to make a more complex shape, we simply add more points like so:

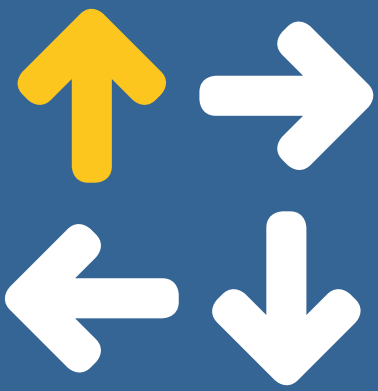
```
while True:

    #pygame.draw.lines(WHERE TO DRAW, COLOUR, CLOSE THE
SHAPE FOR US?, THE POINTS
TO DRAW, LINE WIDTH)

    pygame.draw.lines(window,(255,255,255), True, ((
50, 50), (75, 75), (63, 100), (38, 100), (25, 75)), 1)

    pygame.display.update()
```

There you have it: your very own pentagon. If you want to make a hexagon, an octagon, or even a triacontagon, just add more points – it's that easy. Why not try experimenting with Pygame to produce some interesting pixel art?

The MagPi ESSENTIALS

[CHAPTER TWO]
ANIMATING SHAPES & PATHS

In chapter two, we'll learn how to move shapes around the screen in different directions and patterns, and at different speeds.



In the first chapter, we looked at creating a veriety of shapes in different sizes and colours. Now we're going to be looking at different ways of moving and manipulating those shapes over time. Once we've covered the fundamentals of moving shapes with code, we can then jump into using keyboard and mouse events to control how and when things move in the next chapter. In this tutorial, we wont't be using one single Pygame program. Instead, we have a couple of different code chunks, each demonstrating a different concept. Each complete program will consist of the **top** code, followed by one of the code chunks, and then finished with the **bottom** code. You can use the same file to test the code, or you can create a different file for each chunk; the result will be the same.

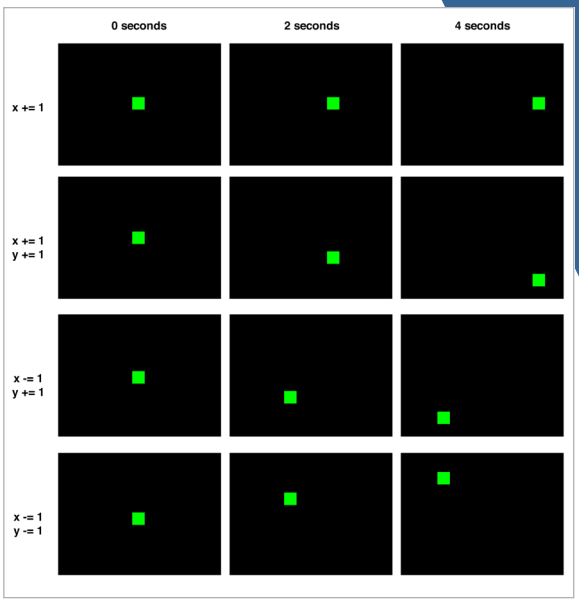
Things to notice

Before we jump into the animation, take a quick look at the import statements on lines 2 and 3 of the **top** code on the page 22. In the last tutorial, we imported all of Pygame and used some of its methods for drawing shapes. That was fine, because we were only drawing static things that didn't take user inputs; from now on, though, we're going to include Pygame's locals and events constants. These are special variables that Pygame includes to help us write more readable code, as well as take away some of the complexity of interacting with the system that we're running our code on. The `pygame.locals` variable mostly contains properties that describe system and game state, so we've called it `GAME_GLOBALS` to reflect this. `pygame.events` includes a list of events, like keyboard events or system events that have happened since Pygame last updated its view; that's why we've imported it as `GAME_EVENTS`. We'll go over exactly what this means exactly in a later chapter; for now, all we're going to use it for in the **bottom** code is to check whether or not our player tried to quit the game as it was running (in this case, by trying to close the window), and then close our program properly.

Below A simulated screenshot showing the random placement of red squares in our window

Moving shapes in time and space

When we think of animation, our minds might turn cartoons and animated films: here, subtle changes in shape and colour trick our brains into seeing movement where there is none. It's no different with computers: whenever you move a mouse or minimise a window, nothing has actually been moved; instead, pixels have been drawn, updated, refreshed, and then drawn again, with everything in its new place.



Above This table demonstrates how different motions affect the position of a shape over time

If you run **chunk 01** (put the **top** code, **chunk 01** code and the **bottom** code together in one file) without uncommenting anything, you'll see a bunch of red squares appearing and disappearing all around the screen. Don't worry, nothing is broken! This is just to demonstrate Pygame drawing, destroying and redrawing things in a window. Add a **#** to the start of the line that starts `surface.fill()`. We use this code to clear the pixel data from the previous frame. Without it, what we see is all of the different frames built up one on top of the other as time passes. `surface.fill()` is like the paint that we use to cover old wallpaper before we add the new one: it creates a blank slate for us to work with.

But that's not very useful, is it? Let's replace **chunk 01** code with **chunk 02** and you'll see a green square moving slowly to the right of the screen.

So, what's making our square move? When we were following the first tutorial, we were drawing shapes like this using numbers that we would pass through to Pygame, like `pygame.draw.rect(surface, (255,0,0), (20, 50, 40, 30))`, and that's all well and good, providing you never want to change anything about that shape. What if we wanted to change the height, width, or colour of this shape? How could we tell Pygame to change the numbers that we've already

[QUICK TIP]

When we run our games, our window is given the title 'Pygame window'. We can set that to any string (series of characters) we like with `pygame.display.set_caption('Pygame Shapes!')`

[QUICK TIP]

If we want to subtract values from a variable, we don't always have to use -= for subtraction and += for addition. We can use += for both; simply add a negative number to take away numbers... e.g. 4 + -3 = 1.

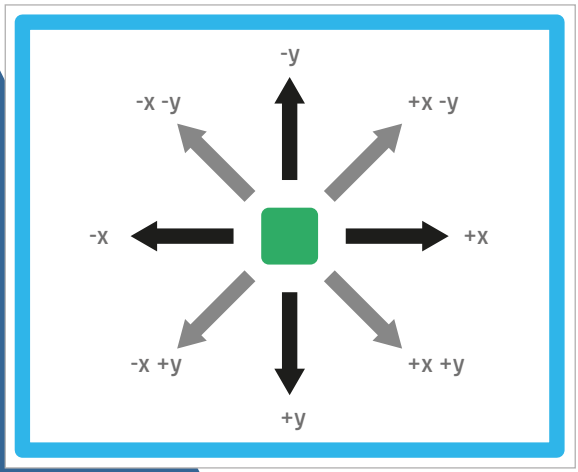
entered? This is where variables come in. Rather than passing through numbers to `pygame.draw.rect()`, we pass in variables instead. After we've drawn the shapes, we can change the variable so that when it's next drawn, it will look slightly different. With **chunk 02**, every time we draw our green square, we add 1 to the variable we use to define its X coordinate (how far it is from the left of the screen), `greenSquareX`. We do this with `+=`, which basically says 'take the current value of the variable and then add whatever number comes after it'.

If we change that line to read `greenSquareX += 5`, every time we draw our square, it will be 5 pixels to the right of where it was the last time it was drawn. This gives the illusion of the shape moving faster than before. If we changed the number we add to `greenSquareX` to 0, our shape would never move; and if we changed it to –5, it would move backwards.

Moving in all directions

So that's how we move left and right; if we can do that much, surely we can go up and down too? Comment out the `greenSquareX` line from **chunk 02** and uncomment the line below by removing the `#`. Our square will start to travel towards the bottom of the screen. Just like before, we're changing the variable that tells our shape where to go, `greenSquareY` (note that we are now changing Y, not X), just a little bit each time to make it move. And, just as we saw by changing the X variable, we can make the green square go up by adding a negative number.

Below This diagram shows the eight directions a shape can move in when using integers



So now we can animate things moving in four directions; that's enough freedom to make so many classic games: *Pokémon*, *Legend Of Zelda*, *Space Invaders*, and more. These games would only move things horizontally and vertically, but never at the same time. The next challenge would be how to make things move diagonally. Fortunately, this is a pretty simple process too.

Download
magpi.cc/1jQielj

TOP

```
import pygame, sys, random
import pygame.locals as GAME_GLOBALS
import pygame.event as GAME_EVENTS

pygame.init()
windowWidth = 640
windowHeight = 480
surface = pygame.display.set_mode((windowWidth, windowHeight))
pygame.display.set_caption('Pygame Shapes!')
```

CHUNK 01

```
while True:
    surface.fill((0,0,0))
    pygame.draw.rect(surface, (255,0,0), (random.randint(
0, windowWidth), random.randint(0, windowHeight), 10, 10))
```

CHUNK 02

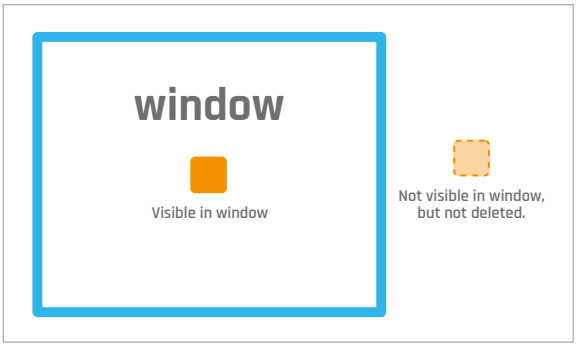
```
greenSquareX = windowWidth / 2
greenSquareY = windowHeight / 2

while True:
    surface.fill((0,0,0))
    pygame.draw.rect(surface, (0, 255, 0),
     (greenSquareX, greenSquareY, 10, 10))
    greenSquareX += 1
    #greenSquareY += 1
    pygame.draw.rect(surface, (0, 0, 255),
       (blueSquareX, blueSquareY, 10, 10))
```

CHUNK 03

```
blueSquareX = 0.0
blueSquareY = 0.0
blueSquareVX = 1
blueSquareVY = 1

while True:
    surface.fill((0,0,0))
    pygame.draw.rect(surface, (0, 0, 255),
       (blueSquareX, blueSquareY, 10, 10))
    blueSquareX += blueSquareVX
    blueSquareY += blueSquareVY
    blueSquareVX += 0.1
    blueSquareVY += 0.1
```

BOTTOM

```
    for event in GAME_EVENTS.get():
        if event.type == GAME_GLOBALS.QUIT:
            pygame.quit()
            sys.exit()
    pygame.display.update()
```



Above The blue box is the viewport of a Pygame window

If we uncomment both `greenSquareX` and `greenSquareY` in our code, then our shape will move to the right and down every time Pygame updates the screen. If we add to our X and Y values, our shapes will move to the right and down. If we add to our X value and subtract from our Y value, then our shapes will move to the right and up. If we subtract from our X value and add to our Y value, our shapes will move to the left and down. Finally, if we subtract from both our X and Y values, our shape will move to the left and upwards. That means we have eight directions that our objects can move in – assuming, that is, that we use numbers that are whole and equal to one another. If we used values that were different for our X and Y values, and we used floats (which are numbers with a decimal place, like 2.3 or 3.141) instead of integers (whole numbers), we could achieve a full 360 degrees of motion.

> If we subtract from our X value and add to our Y value, our shapes will move to the left and down...

Let's play with numbers and decimals a little more. So far, the values we've used to animate our shapes around the screen have been integers that remain constant. With each frame, we would always add 1 (or some other arbitrary value) to move our object. But what happens if we change the values that we use to animate things? What if, instead of adding 1 to X/Y coordinates, we add 1, then 1.1, then 1.2, and so on?

Replace the code from **chunk 02** with the code from **chunk 03** (or create a new file with the **top + chunk 03 + bottom** code). Now if we run that, what do we see? We're adding to both our X and Y values,

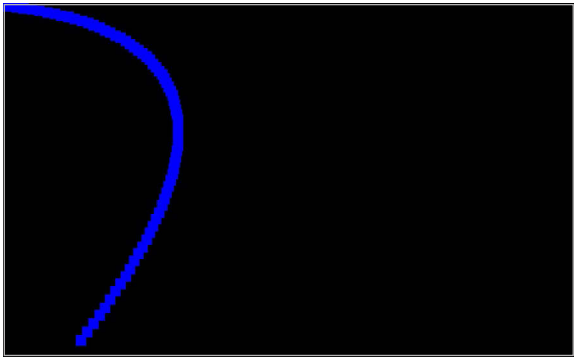
CHUNK 04

```
rectX = windowWidth / 2
rectY = windowHeight / 2
rectWidth = 50
rectHeight = 50

while True:
    surface.fill((0,0,0))
    pygame.draw.rect(surface, (255,255,0), (
rectX-rectWidth /2, rectY-rectHeight /2, rectWidth,rectHeight))
    rectWidth += 1
    rectHeight += 1
```

CHUNK 05

```
squaresRed = random.randint(0, 255)
squaresBlue = random.randint(0, 255)
squaresGreen = random.randint(0, 255)

while True:
    surface.fill((0,0,0))
    pygame.draw.rect(surface, (squaresRed, squaresGreen,
squaresBlue), (50, 50, windowWidth / 2, windowHeight / 2))
    if squaresRed >= 255:
        squaresRed = random.randint(0, 255)
    else:
        squaresRed += 1
    if squaresGreen >= 255:
        squaresGreen = random.randint(0, 255)
    else:
        squaresGreen += 1
    if squaresBlue >= 255:
        squaresBlue = random.randint(0, 255)
    else:
        squaresBlue += 1
```

so our square is moving down and to the right, but something is different from our previous bits of code: as our program continues to run, our square moves to the right a little more than it did in the previous frames. It's accelerating. This is because we're using variables that store a basic measure of speed. By using a variable to add a value to our X and Y coordinates, we can increase the amount of distance that is added in each frame, which gives the illusion of acceleration. If we were to change our code so that it increased our speed variables (**`blueSquareVX`** / **`blueSquareVY`** in this case) through multiplication instead of addition or subtraction, our shapes would accelerate exponentially; we'd have hardly any time to see them before they ran off the screen.



Above This is the path travelled by a shape moving across the window while accelerating

Speaking of which, what happens to our shapes when they run off an edge and are no longer on our screen? Have they disappeared forever? The answer is no. You can think of our window like an actual window in your house. If you look out of the window to see a pedestrian who then moves further down the street so you can no longer see them, they haven't ceased to exist. They're just beyond your line of sight. If our shapes move further across our screen so that we can no longer see them, they don't stop moving or disappear, they keep on going for ever, or until you tell them to stop and come back.

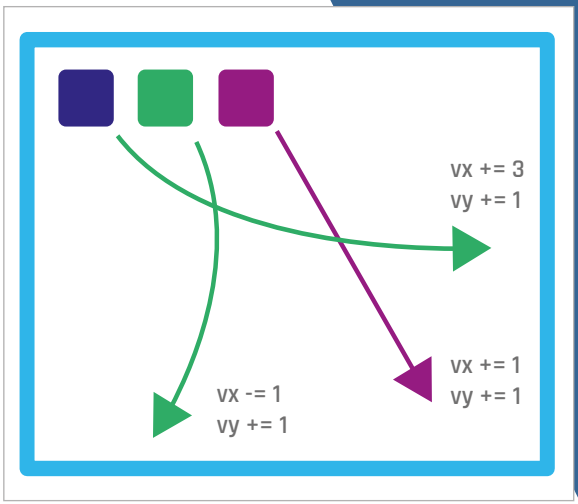
Change the third line in **chunk 03** to read `blueSquareVX = 8`, change the penultimate line in **chunk 03** to `blueSquareVX -= 0.2`, and comment out the last line. Now when we run **chunk 03** for the last time, we see that our square moves to the right across our screen, before slowing to a stop and then coming back on itself, forming an arcing animation. This is because the `blueSquareVX` variable has entered minus numbers, but the `blueSquareVY` variable continues to increase. If we had subtracted the `VX` and `VY` variables in equal values, with equal starting speeds (both `VX` and `VY` being 8, for example), our shapes would have continued along their path, stopped, and then reversed along the exact same path, with the same rate of acceleration as it slowed. Play with theses values to see what effect they have on how our shape moves. If you like, you can comment out the `surface.fill` line and you'll see the path our shape takes trailing behind it.

Animating other properties

Animation isn't just about making things move: it's about making things change, too. Until now, we've been animating shapes by moving them, but we can use the same approach of changing variables over time to affect other properties, like the dimensions of our shapes. For this, switch out the **chunk 03** code for **chunk 04**. Here, `pygame.draw.rect` draws a rectangle just the same as we've done before, but, as in other examples, we've replaced the parameters that determine the

width and height of our rectangle with variables that we change.

We also do a little bit of maths in our code. As the square gets larger, the point from which it is drawn won't change, so the shape will get bigger, but it will do so off-centre from the rest of the window. By subtracting half of the width and half of the height from the coordinates that we draw our shape at, our square will remain in the centre of the window as it gets larger. The nice thing about using variables in our maths is that no matter how we change our variables, the shape created will always be in the centre of the window. Change the number on the `rectWidth` line to any other number between 2 and 10. Now, when our square enlarges, it becomes a rectangle, because its width increases faster than its height does, but it still remains in the centre.



Above This is demonstrating the varying effects of different acceleration values on shapes

The same effect works in the opposite direction. If we start off with a square that has a width and a height of 50, which we can do by setting the variables `rectWidth` and `rectHeight` to 50 and change the `+=` on those lines to `-=`, our square will decrease in size while remaining central to our window.

Something curious happens when our shape reaches a width and height of 0: it starts to grow again. Why? When we hit 0, we start to draw our rectangle with negative numbers, which we are offsetting against with our maths. So, when we draw a shape with a negative width and then offset it against a negative number, our starting points become positive numbers again, albeit mirrored. We can't see the effect because we're using solid colours, but if we were to use the same expanding/shrinking code with an image, it would be flipped upside-down and back-to-front. That's just one of many little quirks that we'll explore in detail later, but for now, we're going to finish up by changing the colours of our shapes over time, by moving onto our last code section, **chunk 05**.

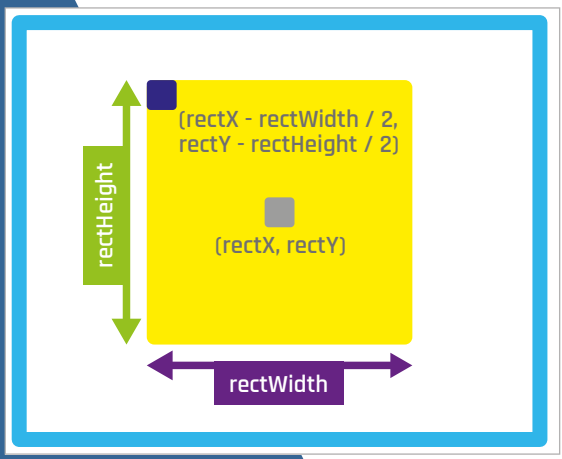
> Games respond via thousands of little checks every few milliseconds

Changing colour over time

Just like our previous pieces of code, we're using variables in place of values to define what our shapes will look like with `pygame.draw.rect`. This code, however, has something a little different from the previous examples. Here, we're not just adding and subtracting values each and every time we draw our shapes; instead, we're checking the values that we have before we change them, using an `if, else` statement.

This is a key concept of game development: how a game responds to a player's actions is a result of hundreds and thousands of these little checks going on every few milliseconds. Without them, there would be no kind of order to any game: it would be like our first bit of code, with the square simply appearing and disappearing at random positions, and there's not much fun in that! With these `if, else` checks, we're making sure that the red, green and blue values never go over 255 (the maximum value that these colours can display at, otherwise Pygame will return an error).

If our values are about to go over 255, we assign them a random value between 0 and 255. The colour of our square will change and will then continue to slowly work its way through the RGB colour palette by adding 1 to our R, G, and B variables (`squaresRed`, `squaresGreen` and `squaresBlue`) as our Pygame program runs. Just as before, if we added a larger number to each of our variables, we would cycle through the available colours more quickly. Similarly, if we added less to each RGB value every time Pygame updates, we would cycle through all of the available colours more slowly. As well as a great learning device, it looks pretty impressive, too.

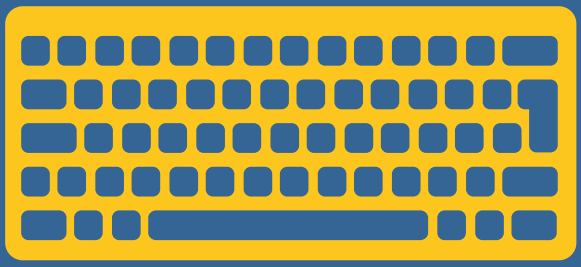


Below Here are the different properties that allow us to centre a square as it enlarges or shrinks

The MagPi ESSENTIALS

[CHAPTER THREE]
TAKING CONTROL OF THE KEYBOARD & MOUSE

In chapter three, we write some code to get to grips with using our keyboard and mouse with Python and Pygame



[QUICK TIP]

Pygame has a set of handy built-in variables for checking which keys are pressed. We've only used a couple, but you can find the complete list at **pygame.org**: **bit.ly/1ycZtzi**

In the first two chapters, we got to grips with the core concepts of drawing and moving shapes of all types, sizes and colours with Pygame. Now that we know our way around Pygame, we're going to start making things that we can play with that are a more interactive. This time, we're going to make two simple programs to learn how to use our keyboard and mouse. For our first program, we will use the keyboard; with it, we'll draw a red square and give it some code so it can move left and right and jump, which may conjure memories of a certain heroic plumber. Our second program will use the mouse. Again, we'll create a square which we can pick up, drag around and which, when we let go of our mouse button, will drop to the floor with the help of a little Pygame-programmed gravity. We're focusing on game dynamics at this point, but don't worry – later chapters will explore the more aesthetic aspects of game design!

So, on to our first program – **keyboard.py**. In contrast to the previous chapter, we're not going to be chopping and changing bits of code to affect the program. If you copy out **keyboard.py** and run it on your Raspberry Pi, it'll run just as we intend it to. This time, we're going to walk through the code line by line to understand exactly what each bit does for the program. Like a lot of things in computing, we are going to start at the top. The first 12 lines of code on page 32 should look pretty familiar to you by now; these are the variables we've used in the previous two parts to define how our window should look, and how we want to interact with Pygame and its methods. The next dozen or so lines are variables that we'll use to determine how our keyboard-controlled square should look and where it should be. Following that, we have two functions, `move()` and `quitGame()`, which we'll use to move and quit the game. Finally, just as in the previous tutorial, we have our main loop where we update our game and redraw all of our pixels.

What keys have we pressed?

How do we know which keys are pressed and when? In the previous chapter, we imported `pygame.events` as `GAME_EVENTS`; now we get to use it. Every Pygame program we write is one big loop that keeps on running forever or until we exit the program. Every time our loop runs, Pygame creates a list of events that have occurred since the last time the loop ran. This includes system events, like a `QUIT` signal; mouse

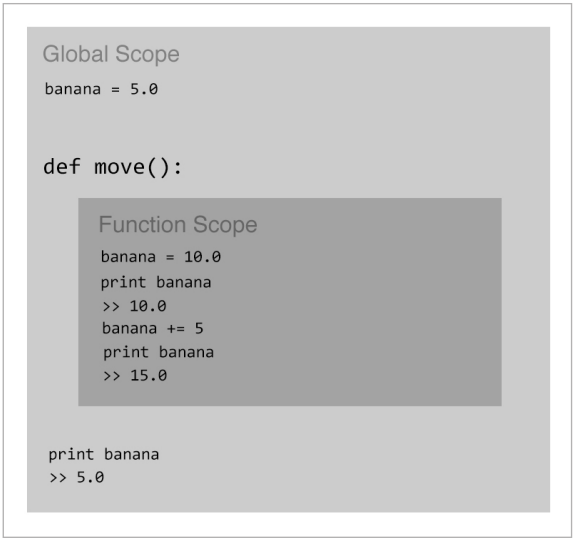
events, such as a left button click; and keyboard events, like when a button is pressed or released. Once we have the list of events that Pygame received, we can decide how our program should respond to those events. If the user tried to quit, we could save the game progress and close the window rather than just exiting the program, or we could move a character every time a key has been pressed. And that's exactly what **keyboard.py** does.

On line 85, we create a **for** loop that will work through each event in the list that Pygame created for us. The events are arranged in the list in the order that Pygame received them. So, for example, if we wanted to use the keyboard events to type in our player's name, we could trust that we would get all of the letters in the right order and not just a random jumble of characters. Now that we have a list of events, we can work through them and check if certain events that are relevant to our game have happened. In **keyboard.py**, we're primarily looking for keyboard events; we can check whether or not an event is a keyboard event by checking its 'type' property with **event.type**. If our event.type is a **pygame.KEYDOWN** event, we know that a key has been pressed; if our event.type is a **pygame.KEYUP** event, we know that a key has

> "Once we have the list of events that Pygame received, we can decide how our program should respond..."

been released. We look for **KEYDOWN** events on line 87 and **KEYUP** events on line 93. We look for **KEYDOWN** events first because logic dictates it: you've got to press a key down before it can pop back up again!

We know have a way of knowing if a key has been pressed, but how do we know which key our player pressed? Every Pygame key event has a 'key' property that describes which key it represents. If we were to print out the **event.key** property, we would see a lot of numbers, but these aren't the keys that we pressed. The numbers we would see are key codes; they're numbers that are uniquely tied to each key on your keyboard, and programmers can use them to check which keys they represent. For example, the **ESC** key on your keyboard is 27, the



Above A basic illustration of code scope

A key is 97, and the **RETURN** key is 13. Does this mean that we have to remember a seemingly disconnected bunch of numbers when we're writing keyboard code? Fortunately, the answer is no. Pygame has a ton of values for checking key codes, which are easier to read and remember when we're writing code. On lines 89, 91, 93, and 97, we use `pygame.K_LEFT`, `pygame.K_RIGHT`, `pygame.K_UP`, and `pygame.K_ESCAPE` to check whether or not any of the key presses are keys that we're looking for.

Once we know that a key has been pressed and which key it was, we can then write code to affect our program in specific ways. For example, if the left arrow key has been pressed, we can move our player to the left with `playerX -= 5`, but we haven't done that here. Why not? Pygame doesn't emit duplicate events for key presses, so if we hold down a key to keep our square moving to the left, nothing would happen. Our square would move the first time Pygame detected the key press, but then it would stop until we pushed the button again. This is intended to help prevent situations where multiple key presses could glitch our games or give a player an unfair advantage, but it doesn't help us very much when it comes to creating games with smooth movement. So how do we get around this? Every time we detect a key press, instead of taking an action, such as moving our square, we set a variable instead. The variables `leftDown`, `rightDown`, and `haveJumped` are the variables that we can use to describe the key states (up or down) to the rest of our program. Whenever we detect that the left arrow button has been pressed, we set `leftDown` to `True`; if we detect that the left arrow button has been released, we set `leftDown` to `False`. If our player holds down the key, `leftDown` will always be `True`, so we can make our Pygame program keep moving our square smoothly across the screen, even though it's not receiving a constant barrage of events telling it to do so.

Keyboard.py

Download
magpi.cc/1jQj5SS

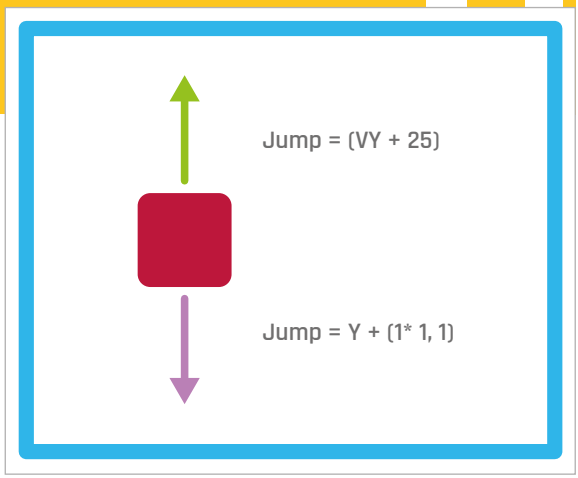
```
01.  import pygame, sys
02.  import pygame.locals as GAME_GLOBALS
03.  import pygame.event as GAME_EVENTS
04.
05.  # Pygame Variables
06.  pygame.init()
07.
08.  windowWidth = 800
09.  windowHeight = 800
10.
11.  surface = pygame.display.set_mode((windowWidth, windowHeight))
12.  pygame.display.set_caption('Pygame Keyboard!')
13.
14.  # Square Variables
15.  playerSize = 20
16.  playerX = (windowWidth / 2) - (playerSize / 2)
17.  playerY = windowHeight - playerSize
18.  playerVX = 1.0
19.  playerVY = 0.0
20.  jumpHeight = 25.0
21.  moveSpeed = 1.0
22.  maxSpeed = 10.0
23.  gravity = 1.0
24.
25.  # Keyboard Variables
26.  leftDown = False
27.  rightDown = False
28.  haveJumped = False
29.
30.  def move():
31.      global playerX, playerY, playerVX, playerVY, haveJumped, gravity
32.
33.      # Move left
34.      if leftDown:
35.          #If we're already moving to the right, reset the
             # moving speed and invert the direction
36.          if playerVX > 0.0:
37.              playerVX = moveSpeed
38.              playerVX = -playerVX
39.          # Make sure our square doesn't leave our
              # window to the left
```

```
        if playerX > 0:
        playerX += playerVX

    # Move right
    if rightDown:
        # If we're already moving to the left, reset
        # the moving speed again
        if playerVX < 0.0:
            playerVX = moveSpeed
        # Make sure our square doesn't leave our
        # window to the right
        if playerX + playerSize < windowWidth:
            playerX += playerVX

    if playerVY > 1.0:
        playerVY = playerVY * 0.9
    else:
        playerVY = 0.0
        haveJumped = False

  # Is our square in the air?
  # Better add some gravity to bring it back down!
    if playerY < windowHeight - playerSize:
        playerY += gravity
        gravity = gravity * 1.1
    else:
        playerY = windowHeight - playerSize
        gravity = 1.0

    playerY -= playerVY

    if (playerVX > 0.0 and playerVX < maxSpeed) or
        (playerVX < 0.0 and playerVX > -maxSpeed):
        if not haveJumped and (leftDown or rightDown)
            playerVX = playerVX * 1.1

# How to quit our program
def quitGame():
    pygame.quit()
    sys.exit()

while True:

    surface.fill((0,0,0))

    pygame.draw.rect(surface, (255,0,0),
```

```
82. (playerX, playerY, playerSize, playerSize))
83.
84.     # Get a list of all events that happened since
        # the last redraw
85.     for event in GAME_EVENTS.get():
86.
87.         if event.type == pygame.KEYDOWN:
88.
89.             if event.key == pygame.K_LEFT:
90.                 leftDown = True
91.             if event.key == pygame.K_RIGHT:
92.                 rightDown = True
93.             if event.key == pygame.K_UP:
94.                 if not haveJumped:
95.                     haveJumped = True
96.                     playerVY += jumpHeight
97.             if event.key == pygame.K_ESCAPE:
98.                 quitGame()
99.
100.        if event.type == pygame.KEYUP:
101.            if event.key == pygame.K_LEFT:
102.                leftDown = False
103.                playerVX = moveSpeed
104.            if event.key == pygame.K_RIGHT:
105.                rightDown = False
106.                playerVX = moveSpeed
107.
108.        if event.type == GAME_GLOBALS.QUIT:
109.            quitGame()
110.
111.    move()
102.
103.    pygame.display.update()
```

Move()

Just after our key detection code we have line 111, which simply has **`move()`** on it. This is a function call. Before now, almost all of the code we've written has been inside our main loop. The problem is that after a while, having every single line of code in one big loop can get a little messy and hard to follow. So, to make our lives easier, we've put the code that's responsible for making our character move into its own function, the **`move`** function. When we call **`move()`**, a lot of code then runs. Let's take a look at what's going on.



Above An example of gravity working against a Y velocity

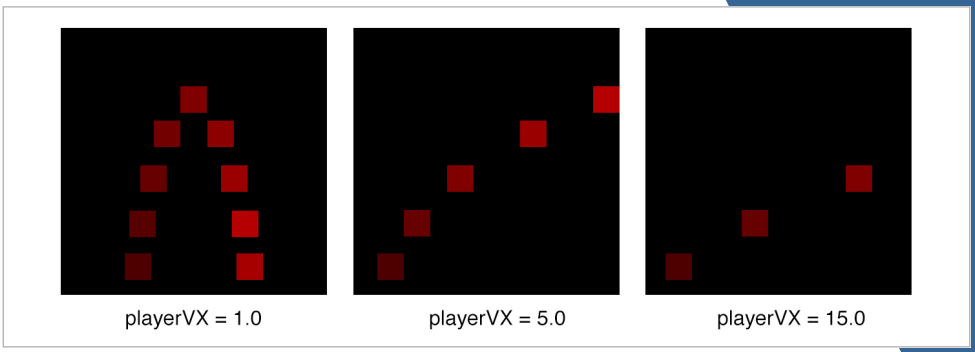
On line 31 we have a `global` statement. Because our code is inside the `move()` function, it no longer has the same scope as our `for` loop. Although we can look at the values of variables outside of our function, we can't set them, unless we include them in the `global` statement. This tells Python that when we call `playerX`, for example, we definitely mean the `playerX` at the top of the file, not a new `playerX` that we might create within the function.

Lines 34 to 50 are where we make our square move left or right, depending on the buttons that have been pressed. If the left arrow button is down, we want to move the square/character/object to the left. This is what we're doing between lines 36 and 41. To do this convincingly, we first need to check whether or not our square is moving already and the direction in which it's going. If our square is already travelling to the right, we need to make it stop and then change direction. Think about it: if you're running in a straight line, you can't turn right around and keep running at the same speed. Rather, you need to stop, turn, and then build the speed up again. Line 37 checks whether our square's X velocity is over 0.0 (going to the right). If it's not, then we either don't need to move at all, or we're already moving to the left, so we can just keep on moving. But if we are moving to the right, setting `playerVX` to `moveSpeed` and then inverting it will stop our square and send it in the correct direction. We don't want our square to run off the screen either; lines 40 and 41 stop our square moving if it's at the left edge of our screen. Lines 44–50 do the exact same thing but in reverse.

Lines 52–70 are a little different. It's here that we add gravity to our square's movement. When we hit the up arrow on our keyboard, our box jumps, but what goes up must come down. Just like when we change direction when we run, we need to slow down after jumping before we start to fall back down again. That's what's going on here.

[QUICK TIP]

The X and Y coordinates of a mouse are relative to the left and top of the window, not the screen that it's in.

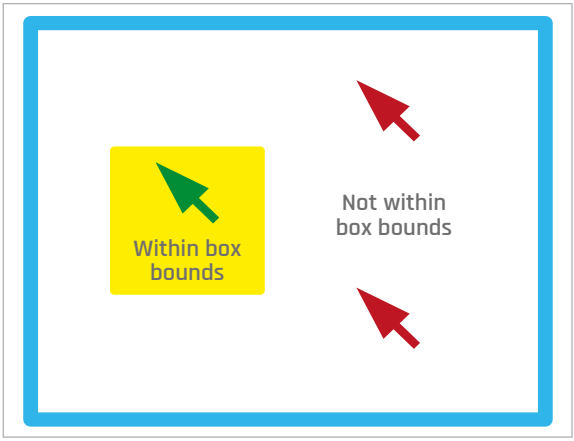


Above A demonstration of the varying effects of the X velocity when jumping

First, on line 52, we check to see whether our square is travelling upwards at a speed greater than 1 pixel per frame. If it is, then we multiply that value by 0.9 so it will eventually come to a point where it is travelling less than 1 pixel per second; when that happens, we set the value to 0 so that we can start falling back to the bottom of the screen. Next, our code checks whether or not our square is in the air: if it is, it will need to come back down. On lines 59–61, we check that the square is in the air and then start adding the **`gravity`** value to the **`playerVY`** value; this will make our square move back down to the bottom of the screen. Each time we add the **`gravity`** value to the **`playerVY`** value, we multiply the former by 1.1; this makes the square speed up as it falls back to the bottom of the screen, just as it would if you threw a ball in the air. Lines 63–64 reset the **`gravity`** and **`playerVY`** values when the bottom of the square hits the bottom of the screen. Lines 68–70 are fun, in that they stop the square from moving any faster left or right once our square has jumped in the air. You can't change direction after you jump; you can only change direction when you hit the ground again, so that's what our square does too.

Pygame mouse events

That's enough of the keyboard for now; it's time for the mouse to shine. The mouse is a simple bit of kit, so the code for it is far less complicated than our keyboard code. If you copy out **`mouse.py`** and run it, you'll see a familiar red square sitting at the bottom of the screen. Pressing your keyboard keys will do nothing this time, for this square is different. If you want to move it, you've got to use the mouse to pick it up. Drag your mouse over the square, hold down the left mouse button



Above An illustration of checking the box bounds against the cursor coordinates

and drag up. Our square moves with our mouse. If you let go of your mouse button, the square will fall back to the bottom of the window. Nice and simple, but how does it work?

This time, we have hardly any code at all in our main `for` loop. Here we're only checking whether or not the first mouse button has been pressed and then we call three functions: `checkBounds()`, `checkGravity()`, and `drawSquare()`. In our `keyboard.py` code, we put some of our code into functions; this time we're doing it to all of them, but we'll get to those in a bit.

The two important things we need to know when using a mouse are where it is and which buttons, if any, have been pressed. Once we know these two things, we can make begin to make things happen. First of all, we're going to find out where the mouse is, and we do that on line 76 with `pygame.mouse.get_pos()`. Unlike our keyboard, we don't have to work through a list of events and check whether they were mouse events. Instead, when we call `pygame.mouse.get_pos()` we get a tuple back with two values: the current X and Y value of the mouse inside the window. Now that we know where the mouse is, all we need to do is determine whether or not any of the buttons have been pressed; we do this on line 81. `pygame.mouse.get_pressed()` returns a tuple of three values: the first is for the left mouse button, the second for the middle mouse button, and the third for the right mouse button. If the button is pressed down, then the value is `True`, otherwise it's `False`. We're not doing anything with the middle or right mouse button, so we can simply check the first value (the left mouse button) with `pygame.mouse.get_pressed()[0]`. If `pygame.mouse.get_pressed()[0]` is `True`, then our player has clicked a button and we can proceed. In this case we set `mousePressed` to `True`, just as we did with `leftDown` and `rightDown` in `keyboard.py`, so we can use it throughout our program.

Checking the square

Now that we know where our mouse is and which buttons are pressed, we can do something with that information. Straight after our code that checks our mouse buttons, we call `checkBounds()` on line 86. `checkBounds()` has one job: to check whether or not our mouse position is within the bounds (edges) of our square. If we were making a fully fledged game, this function would probably check the position of every game object against the mouse coordinates, but in this example we're only interested in our red square. Line 31 checks whether or not our mouse button has been pressed – after all, there's no point in checking where our mouse is if it's not doing anything. If our mouse button has been pressed, on line 33 we look at where the X coordinate of the mouse is and compare it to the X coordinate of our square. If our mouse X is greater than the left of our square and is smaller than the X value of the right of our square (`squareX + squareSize`), we know that our mouse is within the X bounds of our square, but that doesn't mean that it's inside our shape. Before we do anything with our mouse, we need to check that the Y coordinate of our mouse is within our square too, which we do on line 35. If the Y value of

> "there's no point in checking where our mouse is if it's not doing anything..."

our mouse is greater than the top of our shape and less than the bottom of it, then we can be certain that our mouse is somewhere inside of our shape. In **mouse.py**, we've checked the X coordinates and Y coordinates on separate lines – we could have done this in a single line, but it would be quite intimidating line to read, let alone write. Now that we know our mouse is positioned within our square and that we've pressed our mouse button, we can set our `draggingSquare` variable to `True`.

Once `checkBounds()` has done its job, `checkGravity()` gets to work. Just as in **keyboard.py**, `checkGravity()` looks at where our square is in the window: if it's not on the bottom of our window, it will accelerate our square to there. However, it will only do this if we've let go of our mouse button, because we don't want our shape to fall to the ground when we're holding onto it.

Mouse.py

Download
magpi.cc/1jQj5SS

```
01. import pygame, sys
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04.
05. # Pygame Variables
06. pygame.init()
07.
08. windowWidth = 800
09. windowHeight = 800
10.
11. surface = pygame.display.set_mode((windowWidth, windowHeight))
12.
13. pygame.display.set_caption('Pygame Mouse!')
14.
15. # Mouse Variables
16. mousePosition = None
17. mousePressed = False
18.
19. # Square Variables
20. squareSize = 40
21. squareColor = (255, 0, 0)
22. squareX = windowWidth / 2
23. squareY = windowHeight - squareSize
24. draggingSquare = False
25. gravity = 5.0
26.
27. def checkBounds():
28.
29.     global squareColor, squareX, squareY, draggingSquare
30.
31.     if mousePressed == True:
32.         # Is our cursor over the square?
33.         if mousePosition[0] > squareX and
    mousePosition[0] < squareX + squareSize:
34.
35.             if mousePosition[1] > squareY and
    mousePosition[1] < squareY + squareSize:
36.
37.                 draggingSquare = True
38.                 pygame.mouse.set_visible(0)
39.
40.     else:
41.         squareColor = (255,0,0)
```

```
42.         pygame.mouse.set_visible(1)
43.         draggingSquare = False
44.
45. def checkGravity():
46.
47.     global gravity, squareY, squareSize, windowHeight
48.
49.     # Is our square in the air and have we let go of it?
50.     if squareY < windowHeight - squareSize and
    mousePressed == False:
51.         squareY += gravity
52.         gravity = gravity * 1.1
53.     else:
54.         squareY = windowHeight - squareSize
55.         gravity = 5.0
56.
57. def drawSquare():
58.
59.     global squareColor, squareX, squareY, draggingSquare
60.
61.     if draggingSquare == True:
62.
63.         squareColor = (0, 255, 0)
64.         squareX = mousePosition[0] - squareSize / 2
65.         squareY = mousePosition[1] - squareSize / 2
66.
67.     pygame.draw.rect(surface, squareColor, (
    squareX, squareY, squareSize, squareSize))
68.
69. # How to quit our program
70. def quitGame():
71.     pygame.quit()
72.     sys.exit()
73.
74. while True:
75.
76.     mousePosition = pygame.mouse.get_pos()
77.
78.     surface.fill((0,0,0))
79.
80.     # Check whether mouse is pressed down
81.     if pygame.mouse.get_pressed()[0] == True:
82.         mousePressed = True
83.     else:
84.         mousePressed = False
85.
```

```
86.     checkBounds()
87.     checkGravity()
88.     drawSquare()
89.
90.     pygame.display.update()
91.
92.     for event in GAME_EVENTS.get():
93.
94.         if event.type == pygame.KEYDOWN:
95.             if event.key == pygame.K_ESCAPE:
96.                 quitGame()
97.
98.             if event.type == GAME_GLOBALS.QUIT:
99.                 quitGame()
```

Our final function is **`drawSquare()`**: its purpose is easy enough to guess. Based on the adjustments of **`checkBounds()`** and **`checkGravity()`**, **`drawSquare()`** will draw the squarefor us. If our square is being moved around by our mouse, it will draw the square at the mouse coordinates. But if we aren't dragging the square around, it will draw a graceful gravity-driven descent back to the bottom of our window. **`drawSquare()`** has one little trick up its sleeve: as well as affecting the position of our square, it also changes its colour: red when not being dragged and green when being dragged. This code could be useful if, instead of a square, we had a character and we wanted to change its graphic to make it look like it was holding onto our cursor.

What we've learned

We've learned that Pygame creates a list of events that occurred every time the frame is updated, and that we can work through them to check for events that we want to use. We learned that Pygame receives key codes when buttons are pressed, but has a big list of key code events that we can use so we don't have to remember all of the numbers. We learned that we can get mouse events whenever we like, and that we can get coordinates of where the mouse is and which buttons are pressed. We've also learned how to simulate gravity and jumping, and we've made ourselves think about how things move in the real world too. Congratulations! We now have the beginnings of a real game.

The MagPi ESSENTIALS

[CHAPTER FOUR]
YOUR FIRST GAME

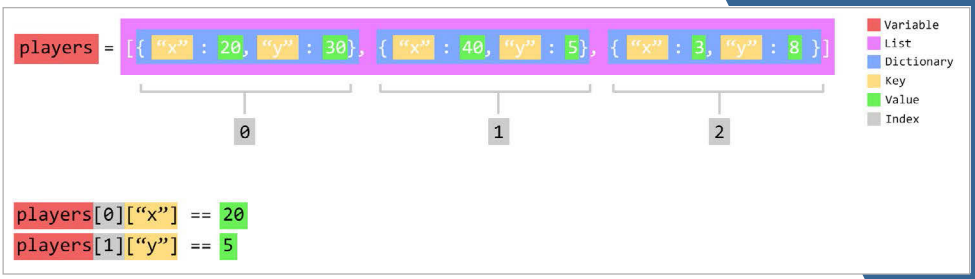
Now that we've covered making shapes, animating them, and setting up control mechanisms, we have everything we need to make our first proper game. We're going to make an old-school drop-down game where platforms rise up from the floor and try to crush our player against the roof; the only way to survive is by dropping through the gaps in the platforms. Unlike our previous examples, we're not going to write a program that just runs: we will also make a simple start screen and a game over screen. We still have a couple of new things we're going to learn about along the way, like loading images and timing events. This is by far the largest piece of code we will have written, but don't worry: if you've followed along so far, you'll recognise much of it already!

How does our game work?

Before we write any code, though, it's important to have a solid understanding of how our game is going to work. When the game starts, our avatar (a red rectangle) will drop down from the top of the screen. Every two seconds, a white platform will start to rise from the bottom of the screen; if our character lands on one of these platforms, it will start to rise along with it. If we go off the top of the game screen, it's game over. Defeat is not assured, however: we can move our character with the left and right arrow keys so it can drop down through the randomly positioned gaps in the platforms. The aim of the game is to stay alive as long as possible. It sounds easy, but things get tougher as time goes on, because the platforms will start to appear after a shorter delay.

Variables and prerequisites

Lines 1–39 of our code (see page 51) contain the `import` statements and variables we're going to need to get our game off the ground. By now, much of this should look pretty familiar. At the top we have our `import` statements, which let us include modules to help with our game development. Take note of the `GAME_TIME` import, as this will be used quite a bit later on. Lines 8 and 9 are loading images that we'll be using for our start and game over screens. We could draw the graphical user interface (GUI) with code, but by using images we're saving ourselves time and effort at the cost of just a few kilobytes.



Above
A handy breakdown of the various components of a dictionary

We've seen lines 11–15 before; these are the variables that we'll use to control how the game window looks. Lines 20–38 are the variables that we'll use to keep track of our game state, like where things are, how big they are, and how they should move. Two of these might jump out as a little bit different from variables we've used previously: **`gamePlatforms`** and **`player`**. One of these looks empty and the other has multiple values. **`gamePlatforms`** is a variable type known as a list, while **`player`** is a type known as a dictionary, and they don't work like other variables do. Let's take a little time to understand how they work.

Dictionaries and lists

In previous chapters, we've almost always used variables that have one value, but there are other variables that can contain multiple values – like tuples, for example – and these are very useful to us as we start to make bigger and more powerful programs. When we write small programs, having variables with a single value is great, because we can see which variables are doing what. However, as programs grow, it can get harder to name variables in a way that relates to what we're trying to do. Let's imagine a game where there's more than one player, like an MMO; if we wrote code like we've done before, we'd need to create multiple sets of variables for each player. It doesn't take a genius to realise the code is going to get unmanageably long, very quickly.

What if we wanted to handle four or 100 or 1,000 players at the same time? Do we hand-write variables for every single one? No. We can use dictionaries and lists instead.

The **`player`** variable on lines 32–38 is a dictionary. A dictionary is a variable with multiple keys that have values. You can think of a dictionary as you would its real-world counterpart: if you want to know what something is, you search through until you find the definition.

> Dictionaries are really useful, because they let us group values together...

So, let's say we want to know what the value of `x` in the `player` dictionary is; all we have to do is request `player["x"]`. We can do the same with any other value that is stored in it, and we can also save or change values.

If the value `player["y"]` is 20 and we wanted to change it to 25, we'd enter `player["y"] = 25`, just like setting any other variable. Dictionaries are really useful, because they let us group values together in a way that's easy to read and easy to access with code. If we revisit our MMO game thought exercise, we'd quickly realise that we'd still need 100 variables to handle 100 players, even though we've made things tidier and more usable. What's the best way to keep track of dictionaries? That's where lists come in.

Lists are variables that can store groups of other variables. If we wanted to keep track of the players in our game, we wouldn't need to make a variable for each player. We could just add a player dictionary to a list and work through them whenever we need to. If, for example, we wanted to get the information for the second player in our imaginary MMO, we'd enter something like `players[1]` and that would return a dictionary for our second player which we could then get values from, like so: `players[1]["x"]`. The `1` in this example is called an index. It's important to notice that list indexes start counting from 0, so if we want to access the first item in a list, we use the index 0; if we want to get the fourth item from a list, we use the index 3.

Below We could code our title screen, but using an image is much simpler

In our game, we're not using lists and dictionaries to track players, but to track the platforms that we'll be moving along and dropping through. We'll have a look at that in a little while, once we've examined the game's logic.

The 'main' game loop

Lines 40–146 are where the logic for our game lives, but the state of our game is controlled in our main loop between lines 149 and 199. Just like our previous programs, on lines 153–176 we listen for various in-game events in our main loop and effect changes based on the events dispatched by our Raspberry Pi (keyboard, exit events, etc). Lines 178–196 are where the state of our game is determined and functions are called accordingly.

Each function between lines 40 and 146 is coded to handle a single aspect of the gameplay independently of the other functions, but they need to be called in a certain order to make sure that our game runs as expected. In order to understand each function and aspect of our game, we're going to work through them in the order that our main loop calls them. When our game first runs, we want to display our welcome screen telling people to press space to start; when the user presses the space bar, we want to start the game and when the user is pushed off the screen, we want to show the game over screen and let them restart. All of this is handled in the main loop. Let's break it down.

The start game screen

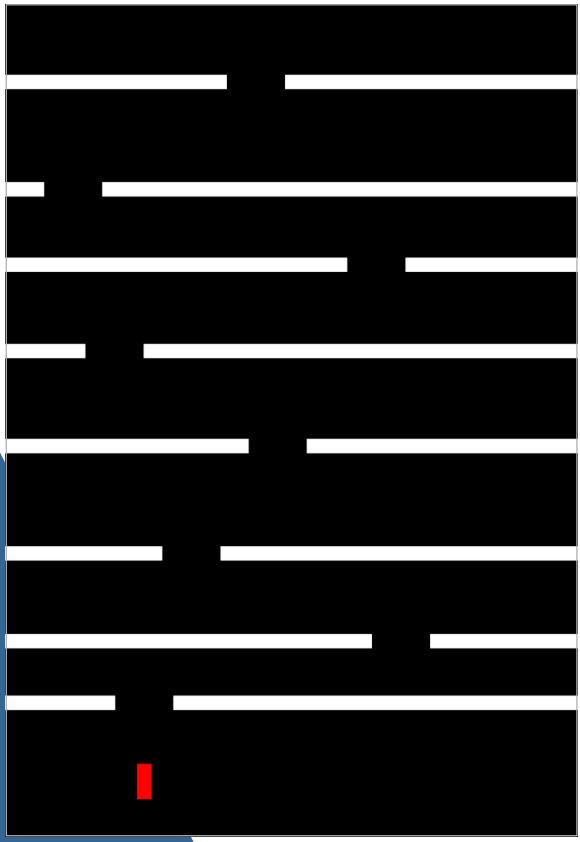
When our game starts, we're presented with the start game screen. This screen is a simple image that we loaded on line 8 of our code listing. At the other end of our listing, on line 189, we draw that image onto our surface. It's in the final `if-elif` statement of our main loop. On line 178, our script checks whether or not a game is already underway; if it is, it will check the game and render the screen as required. If there isn't a game underway, the loop then checks whether or not a game has ended on line 187, in which case we want to display the player's score on the game over screen and offer them the option to play again. If a game has neither been started nor finished, we can infer that we've just started the game and so we can render the start screen.

In order to start the game, we're checking for a space bar keyboard press on line 170. If a game hasn't been started (or has just finished), the `restartGame` function on lines 133–142 is called. All this function does is reset all of the variables for a new game. On the next loop of our main loop, the settings will be in place for a new game and one will be started.

The game platforms

Once the space bar has been pressed, a new game will begin. All of the game logic is between lines 40 and 146, but we call the methods to generate the game on lines 182–185 in a particular order. First, we call `movePlatforms()`, which works through every platform in the game and moves it up the screen at the speed set with the variable `platformSpeed`. `movePlatforms` also checks whether or not the platform has reached the top of our game window; if it has, it will remove that platform from our `gamePlatforms` list. You may notice that the `for` loop on line 110 is a little different from those we've used in the past. Unlike most `for` loops in Python, this one passes the index through to the loop with the `idx` value. We need this index so we can remove the right platform from the `gamePlatforms` list, otherwise we'd have to work through the list and try to figure out which one needs to go each time, and that wouldn't be good for the frame rate. The function `pop` removes an item from a list at a given point; if we wanted to remove the second platform in the list, for example, we'd pass `gamePlatforms.pop(1)` – remember, lists begin at 0, so 1 is the second item in our list.

Below Our 'Drop' game starts off easy, but gets harder as it progresses, creating platforms more quickly than at first



Once we've worked out where the platforms need to go and which ones need to go away, we can draw them. We do this with `drawPlatforms` on lines 118–123. Nothing fancy here; we're just drawing a white rectangle which is the width of the screen, and then a black rectangle for the gap that the character can drop through to the next platform.

But where do these platforms come from? On line 196, we find the answer. Pygame keeps track of how long the game has been running for with its `get_ticks` function. We want to release a platform around every 2 seconds; so, on each loop, we check to see how long it has been since we created a new platform by subtracting the time that we last created a platform from the current game time, which we access with `GAME_TIME.get_ticks()`. The game time is recorded in milliseconds, so if 2,000 milliseconds (1,000 milliseconds = 1 second) have passed since we generated a platform, it's time to create a new one; we do that with `createPlatform` on line 94.

On line 96, we use `global` to tell our code that we want to use variables that exist in the global scope, not create new ones with the same name in the function scope. On lines 98–99 we're creating variables that will define the position of the platform (`platformY`) and the location of the gap through which to drop along with it (`gapPosition`). We want our platform to always rise from the bottom of the window, but the gap can be at any point along the length of the platform.

Just as tracking players becomes difficult when we have lots of them to deal with, the same is true of our platforms here. We're generating a platform every 2 seconds, and that delay gets smaller each time. If you've been playing for more than a minute, you'll have jumped on something like 100 platforms! We can't pre-program all of those and even if we could, our game would become very bland after a couple of plays. Line 101 is where we create our new platforms. Like any list, we can add new items to it; in Python, we can do this with `.append()`. In this case, we're creating a dictionary with everything we need to create a platform, the position of the platform (stored with the `pos` key), and the location of the gap (stored with the `gap` key).

Moving our avatar

Our avatar isn't a complicated construct: at its simplest, it's a red rectangle. After our platforms are drawn, we want to work out where our avatar can go. As you'll see, the `movePlayer()` function on lines 44–92 is home to most of our game's logic. Before we look at the code, let's talk about what's happening: we want our player to fall when there is either a gap in the platform or no platform at all. We also want the avatar to travel up with the platform if there is no gap present. To code

```
gamePlatforms.append({"pos" : [0, platformY], "gap" : gapPosition})
```

List
Dictionary
Key
Value

Above Here's a handy reference to help with appending a dictionary item to a list

this logic, we could check the position of all of the platforms every frame and write some code that would figure out whether or not our avatar is on top of a platform or not, but that's really going to make our Pi work hard and wouldn't be efficient. Instead, we're doing something simpler: our platforms are always white and our background is always black, so if we can know the colour of the pixel just beneath our avatar, we can work out whether or not we need to drop or not.

We also need to check that our avatar is completely off the edge of our platform before we drop. To do this, we check the values just beneath our avatar, to both the left and the right. We can get the colour of a pixel at a certain point with `surface.get_at((X, Y))`; this will return a tuple with four values `(RED, GREEN, BLUE, OPACITY)`, each between 0 and 255, just as if we had set the colours ourselves. On lines 51–52 we check the colour beneath the bottom left of our avatar, and on lines 54–55 we do the same for the bottom right. If the colour values we find at either the bottom left or the bottom right of the avatar are `(255,255,255,255)` (white), then we know at least one edge of our avatar is still on a platform. If both are anything but white, then there's a gap in the platform or we're in blank space, so we can let our avatar drop. This all happens on lines 57–68. We also check that we don't let our avatar run off the bottom of our window.

So, that's the code that handles what to do if we aren't on top of a platform, but what about when we want our avatar to travel with the platform? If our avatar finds itself unable to go down, we need to work out where the platform stops and the blank space starts. We do this on lines 64–80. On lines 66 and 67 we set two variables, `foundPlatformTop` and `yOffset`; we use these values to help our `while` loop on lines 70–80. When we find a white pixel beneath either the bottom left or right of our avatar, we need to work backwards to move our avatar up with the platform. Our `while` loop subtracts 1 from our `player["y"]` value and checks the colour that it finds there. Remember, we haven't drawn our avatar yet, so the only colours on our surface are black (background) or white (platforms). If the coordinates checked are white, 1 is added to the

`yOffset` and the `while` loop continues to search for a black pixel. It will do this until it finds a black pixel above the x coordinate of our avatar, adding 1 to the `yOffset` variable each time. Once a black pixel is found, we've discovered where our platform ends and can subtract the `yOffset` from `player["y"]` to put our avatar just on top of the platform; this is done on line 73. If we don't find a black pixel before we reach the top of the surface, it's game over: our avatar is trapped off screen.

Moving our character left and right is done on lines 82–92. If the code looks familiar, it's because we used it in our last tutorial to move our squares around. Now that we've worked out where our avatar can go, we can draw it by calling `drawPlayer()` on line 203.

Game over

We're almost done; the last thing we want to handle is what happens when our player loses the game. The game is over once our avatar disappears off the top of our screen, and we want to tell the user that. When our avatar disappears, we call the `gameOver` function on line 79. All the `gameOver` function does is set some variables that our main loop will check to see if the game is underway. Once `gameEnded` is `True` and `gameStarted` is `False`, our main loop will draw our game over screen. Just like our welcome screen, we draw our game over image onto the surface on line 193 and give the player the option to restart the game with another space bar press.

And that's it! Using all the skills we've already acquired (and a few new ones), we've built our first fully fledged game. Like all good games, we've got a start, a middle, and an end.

Left Just like our start screen, our game over screen is simply an image drawn straight onto our surface when we need it

Just_drop.py

Download
magpi.cc/1jQj9Cb

```
01. import pygame, sys, random
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05.
06. pygame.init()
07.
08. title_image = pygame.image.load("assets/title.jpg")
09. game_over_image = pygame.image.load("assets/game_over.jpg")
10.
11. windowWidth = 400
12. windowHeight = 600
13.
14. surface = pygame.display.set_mode((windowWidth,
    windowHeight))
15. pygame.display.set_caption("Drop!")
16.
17. leftDown = False
18. rightDown = False
19.
20. gameStarted = False
21. gameEnded = False
22. gamePlatforms = []
23. platformSpeed = 3
24. platformDelay = 2000
25. lastPlatform = 0
26. platformsDroppedThrough = -1
27. dropping = False
28.
29. gameBeganAt = 0
30. timer = 0
31.
32. player = {
33.   "x" : windowWidth / 2,
34.   "y" : 0,
35.   "height" : 25,
36.   "width" : 10,
37.   "vy" : 5
38. }
39.
```

```
40. def drawPlayer():
41.
42.   pygame.draw.rect(surface, (255,0,0), (player["x"],
    player["y"], player["width"], player["height"]))
43.
44. def movePlayer():
45.
46.   global platformsDroppedThrough, dropping
47.
48.   leftOfPlayerOnPlatform = True
49.   rightOfPlayerOnPlatform = True
50.
51.   if surface.get_at((player["x"], player["y"] +
    player["height"])) == (0,0,0,255):
52.     leftOfPlayerOnPlatform = False
53.
54.   if surface.get_at((player["x"] + player["width"], player[
    "y"] + player["height"])) == (0,0,0,255):
55.     rightOfPlayerOnPlatform = False
56.
57.   if leftOfPlayerOnPlatform is False and
    rightOfPlayerOnPlatform is False and (
    player["y"] + player["height"]) + player["vy"] < windowHeight:
58.     player["y"] += player["vy"]
59.
60.     if dropping is False:
61.       dropping = True
62.       platformsDroppedThrough += 1
63.
64.   else :
65.
66.     foundPlatformTop = False
67.     yOffset = 0
68.     dropping = False
69.
70.     while foundPlatformTop is False:
71.
72.       if surface.get_at((player["x"], (
    player["y"] + player["height"]) - yOffset )) == (0,0,0,255):
73.         player["y"] -= yOffset
74.         foundPlatformTop = True
75.       elif (player["y"] + player["height"]) - yOffset > 0:
76.         yOffset += 1
```

```
77.         else :
78.
79.           gameOver()
80.           break
81.
82.     if leftDown is True:
83.       if player["x"] > 0 and player["x"] - 5 > 0:
84.         player["x"] -= 5
85.       elif player["x"] > 0 and player["x"] - 5 < 0:
86.         player["x"] = 0
87.
88.     if rightDown is True:
89.       if player["x"] + player["width"] < windowWidth and (
    player["x"] + player["width"]) + 5 < windowWidth:
90.         player["x"] += 5
91.       elif player["x"] + player["width"] < windowWidth and (
    player["x"] + player["width"]) + 5 > windowWidth:
92.         player["x"] = windowWidth - player["width"]
93.
94.   def createPlatform():
95.
96.     global lastPlatform, platformDelay
97.
98.     platformY = windowHeight
99.     gapPosition = random.randint(0, windowWidth - 40)
100.
101.    gamePlatforms.append({"pos" : [0, platformY],
    "gap" : gapPosition})
102.    lastPlatform = GAME_TIME.get_ticks()
103.
104.    if platformDelay > 800:
105.      platformDelay -= 50
106.
107.  def movePlatforms():
108.    # print("Platforms")
109.
110.    for idx, platform in enumerate(gamePlatforms):
111.
112.      platform["pos"][1] -= platformSpeed
113.
114.      if platform["pos"][1] < -10:
115.        gamePlatforms.pop(idx)
116.
```

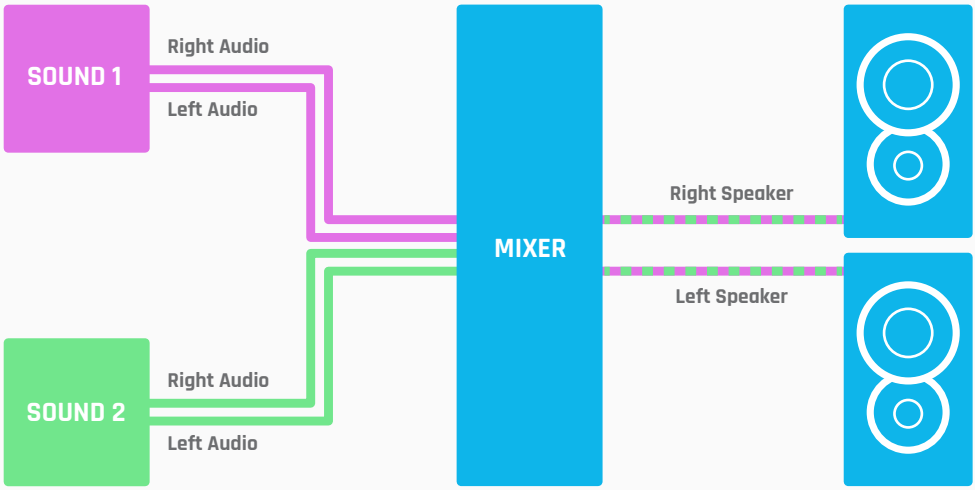
```
117.
118. def drawPlatforms():
119.
120.   for platform in gamePlatforms:
121.
122.     pygame.draw.rect(surface, (255,255,255), (platform["pos"]
     [0], platform["pos"][1], windowWidth, 10))
123.     pygame.draw.rect(surface, (0,0,0), (platform["gap"],
     platform["pos"][1], 40, 10) )
124.
125.
126. def gameOver():
127.   global gameStarted, gameEnded
128.
129.   platformSpeed = 0
130.   gameStarted = False
131.   gameEnded = True
132.
133. def restartGame():
134.
135.   global gamePlatforms, player, gameBeganAt,
     platformsDroppedThrough, platformDelay
136.
137.   gamePlatforms = []
138.   player["x"] = windowWidth / 2
139.   player["y"] = 0
140.   gameBeganAt = GAME_TIME.get_ticks()
141.   platformsDroppedThrough = -1
142.   platformDelay = 2000
143.
144. def quitGame():
145.   pygame.quit()
146.   sys.exit()
147.
148. # 'main' loop
149. while True:
150.
151.   surface.fill((0,0,0))
152.
153.   for event in GAME_EVENTS.get():
154.
155.     if event.type == pygame.KEYDOWN:
156.
```

```
157.      if event.key == pygame.K_LEFT:
158.        leftDown = True
159.      if event.key == pygame.K_RIGHT:
160.        rightDown = True
161.      if event.key == pygame.K_ESCAPE:
162.        quitGame()
163.
164.    if event.type == pygame.KEYUP:
165.      if event.key == pygame.K_LEFT:
166.        leftDown = False
167.      if event.key == pygame.K_RIGHT:
168.        rightDown = False
169.
170.      if event.key == pygame.K_SPACE:
171.        if gameStarted == False:
172.          restartGame()
173.          gameStarted = True
174.
175.    if event.type == GAME_GLOBALS.QUIT:
176.      quitGame()
177.
178.    if gameStarted is True: # Play game
179.
180.      timer = GAME_TIME.get_ticks() - gameBeganAt
181.
182.      movePlatforms()
183.      drawPlatforms()
184.      movePlayer()
185.      drawPlayer()
186.
187.    elif gameEnded is True:
188.      # Draw game over screen
189.      surface.blit(game_over_image, (0, 150))
190.
191.    else :
192.      # Welcome Screen
193.      surface.blit(title_image, (0, 150))
194.
195.    if GAME_TIME.get_ticks() - lastPlatform > platformDelay:
196.      createPlatform()
197.
198.    pygame.display.update()
199.
```

The MagPi ESSENTIALS

[CHAPTER FIVE]
PYGAME SOUNDBOARD

In chapter five, we learn about loading and playing sounds in your Pygame projects by making a fun farmyard soundboard.



Above A basic diagram of how the Pygame audio mixer works

In the previous chapter, we put together a simple video game in which we tried to avoid the dreadful fate of being crushed by a ceiling by dropping through platforms into the space below. It didn't have the fanciest graphics, but, then again, fancy graphics aren't everything. One simple thing that we can do to enhance our players' experience is to add sounds, and that's what we're going to be doing here. We're going to learn how sounds work with Pygame by putting together a soundboard with some simple controls. We'll learn about loading sounds, playing them, adjusting the sound controls, and using the mixer to stop everything. We'll also put together some code to create the soundboard buttons; this will draw from our knowledge of lists, dictionaries, and mouse events which we have gained in previous chapters.

While MP3 is a really popular format for playing music and sounds (no doubt you have thousands of files sitting on a hard drive somewhere), the downside is that it's a proprietary technology. As such, Pygame and other popular libraries don't support MP3 out of the box, perhaps because they can't afford to pay for a licence. We, therefore, are going to use OGG, an open sound format that your Pi and Pygame will play without any problems at all. All of the sounds for this project are available on GitHub, in OGG and MP3 format, for you to play with. You can download the code and sounds here: **bit.ly/1J7Ds6m**.

First things first

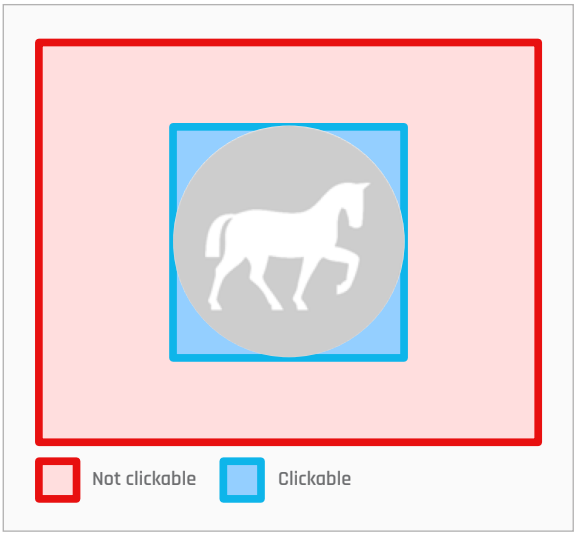
Just like any Pygame project, there are a couple of things we need to sort out before we can get our hands dirty writing some real code. Lines 1–14 should look really familiar to you by now: first we have our import statements on lines 1–5 , then we set the properties of our windows on lines 6–11, and finally we create a couple of variables for use in our Pygame program a little later on lines 13–17. If you look at line 13, you'll see the `buttons` variable; when we're ready to start creating our buttons, we'll append some dictionaries to this list so we can easily keep track of all of the soundboard buttons we create. On the next line, we have our `stopButton` dictionary; when we create our stop button, it'll behave much like the rest of the buttons on our soundboard except that it will stop all current sounds playing. Since it's unique, our stop button gets its own variable.

> "The sound object fits the bill for our soundboard better..."

On lines 83–106 we have our familiar old 'main' loop. It's looking a lot smaller than last time: that's because we've broken out all of the code that we could put in main into separate functions. If we didn't, things would start to get quite messy and hard to follow. By having functions that handle one thing very well, we can write a program that runs well and looks great, too. Just as before, our main loop is responsible for wiping the screen (line 84); handling mouse, keyboard, and system events (lines 88–100); and calling functions to draw in our window.

Let's mix it up with Pygame mixer

If you're going to use sounds in Pygame, you're more than likely going to be using Pygame's built-in mixer. You can think of the mixer like its real-world equivalent: all sounds across the system (or in our case, across hte game) pass through it. When a sound is in the mixer, it can be adjusted in a variety of ways, volume being one. When our mixer is finished, it passes the sound through to an output, which, in this case, is our speakers. So, before we start loading or playing any sounds, we need to initialise the mixer, just as we need to initialise Pygame before we draw things; we do that on line 19.



Above A diagram visualising the imaginary bounding box that surrounds the buttons in our Pygame program

Our first sound

You can play sounds a couple of different ways in Pygame: you can either play a stream of sound, which you can think of as sound being played as it's being loaded, or you can create and play a sound object, which loads the sound, stores it in our Raspberry Pi's memory, and then plays it. Each way of playing sound is good for different instances. The streaming of sound is better, for example, when we want to create background music that plays while we are doing other things, whereas the sound object is a better choice for when we want to play short sounds quickly and often.

The sound object fits the bill for our soundboard better than the sound stream, so we'll use those for our buttons a little later on. First we're going to add some ambience to our soundboard with a little background audio from a farm. Background audio usually loops without any sort of user interaction, and streaming audio can be set to loop without too much trouble, so that's what we're going to do. Before we can play any music, we need to load it: on line 20 of our program we point Pygame to our background audio **farm.ogg**. This loads the audio into our mixer, but it won't play straight away. On line 21 we call **`pygame.mixer.music.play(-1)`**, which starts playing our sound file. The number we pass is the number of times we want to our sound to repeat before it stops playing. We've passed -1, which means that it will loop forever, or until we stop it. If we ran our soundboard at this point, we'd have a lovely big blank window with some calming farm sounds playing, but that's a bit bare. It's time to make some buttons!

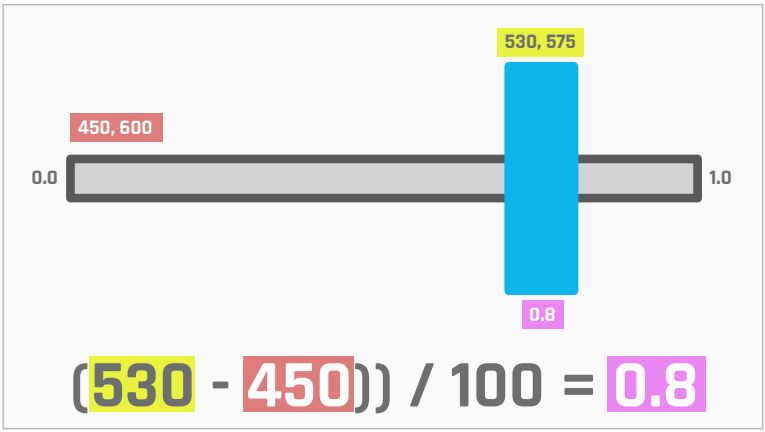
A button here, a button there, buttons EVERYWHERE!

So, how are we going to make these buttons? We could do what we've done in previous chapters and draw some shapes and add text to them; that would certainly do the job, but it won't look great. Instead, we're

going to make our buttons out of some images your expert has put together for each different animal sound. If you want to take a peek at the buttons before loading them, they're included in the folder **code/assets/images**, which you can grab from the GitHub repo. Each button has a silhouette of an animal. It will make the sound this animal makes when we click it, but how do we make an image make a sound? We are going to be using lists and dictionaries again: remember the **buttons** variable we looked at right at the start of this chapter? You can see that it's currently empty, but now it's time to add some dictionaries describing our buttons to it.If you look at lines 71–80, you'll see that each line creates a new dictionary for each animal. Each dictionary has three keys (or properties: the terms are interchangeable). The first one is **image**, which will load the image for that button for us. In previous dictionaries, we've stored strings in dictionaries and then used those strings to load images when we've needed them; this time, however, we've actually loaded each image into our dictionary with **pygame.image.load()**. This saves time when we have to draw something many times, and seeing as

> "The last property, sound, is just like our image property, except that it loads a sound"

the image never changes, it makes sense to have it there. Our next key is **position**; this is a simple tuple that contains the x and y coordinates for where our buttons will be drawn. The last property, **sound**, is just like our image property, except that, as you might expect, it loads a sound instead of an image. Here we're loading the sounds as objects, which means that they're essentially self-contained in terms of how they work. With the background audio we loaded earlier, we passed the data straight through into the mixer and played it through the latter. A sound object, however, has functions that let us control the audio by itself. For example, we could call **sound.play()** and the sound would play, or we could call **sound.stop()**, but it would only apply to the sound we were calling those functions on: if we had two sounds playing at the same time and we stopped only one, the other would keep playing.



Right An illustration of the equation used to control the volume

Drawing our buttons

On lines 71–80, we've added nine different buttons, but if we were to run our program without finishing it, we would still only see a blank white window. This is because we haven't drawn the buttons yet. We've only loaded the necessary code to make them work. In our main loop on line 101, we call the function `drawButtons()`, which can be found on lines 23–28; this will draw the buttons to our surface. You may be surprised that we can use such a small amount of code to draw nine buttons. This is due to the fact that we've done all of the hard work of finding and loading our images and sounds before our main loop could even run: we have very little to do when we actually draw the buttons to our surface. On line 25 we have a `for` loop which works through the buttons list we looked right at the start on line 13; for every dictionary it finds in the list, it will draw a button onto our surface, using the properties it finds and a process called blitting. This happens on line 26. Blitting is something you might have come across in the past, but don't worry if you haven't: it's essentially a fancy way of saying 'paste', and we used it in our last tutorial to draw the start and finish screens for our drop game. When we blit something, we take the pixels of our surface and then we change the pixels so that they're the same as the image we're adding. This means that anything that was beneath the area being blitted is lost. It's a lot like cutting letters out of a newspaper and supergluing them onto a piece of paper: whatever was beneath the newspaper clipping is gone forever, but we can see what we cut out of the newspaper just fine.

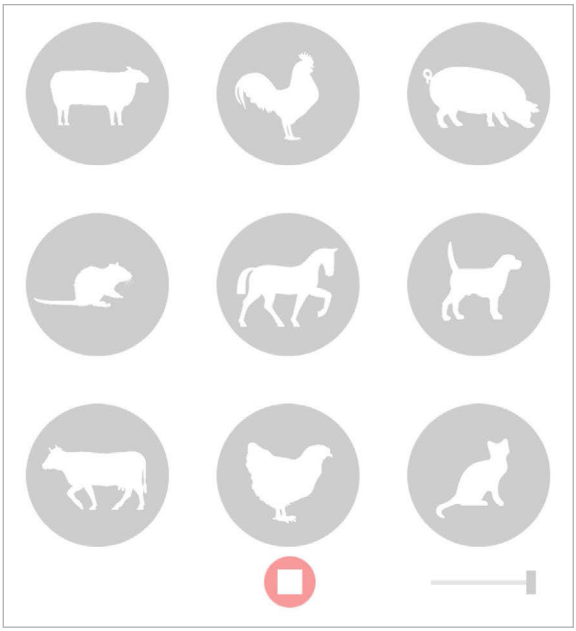
Clicking our buttons

Now that we have a soundboard with buttons, we need to make those buttons do something. Buttons are one of those things that can be really simple or really tricky: some systems do a lot of the work for you, whereas others don't do so much. Unfortunately, Pygame is one of the latter systems, but that doesn't matter: we can write the code for our buttons ourselves. On lines 89–100 we have code which handles some of the events that happen in Pygame. You'll recognise the code on lines 89–97: it's the same code that we've used to quit Pygame for the last four tutorials, but on lines 99–101 you'll notice we are looking for a `MOUSEBUTTONUP` event. If you are wondering why we look for `MOUSEBUTTONUP` and not something like `MOUSECLICK`, remember that

> "Buttons are one of those things that can be really simple or really tricky..."

for a mouse button to go up, it has to have gone down first, meaning that the mouse must have been clicked. If the mouse has been clicked, we call the `handleClick()` function, which is on lines 38–55. Just like when we drew our buttons, we're going to work through the buttons list to find out where they are on our surface. If our mouse clicked where a button is, we'll play that sound, otherwise we'll do nothing.

In order to check whether or not a button was clicked, we need to know three things: 1) the position of each button, 2) the size of that button, and 3) where the mouse was when it was clicked. If our mouse coordinates are greater than the x and y coordinates of the button image, but less than the x and y coordinates plus the width and height of the image, then we can be sure that the button we're checking against was clicked and we can therefore play the sound for that button; otherwise, the mouse was outside the button. Checking this way is a little bit of a cheat: our buttons are circles, but we check whether or not a click has happened within a square that surrounds



Above A screenshot of our finished soundboard

the button. We do this because the result is almost exactly the same and the code to check for a click in a square is much quicker than that for checking a circle or irregular shape. This square is often referred to as a bounding box, and it's often used to check for clicks.

The checks happen on lines 47 and 49. If either statement is found to be False, then nothing will happen, but if both are correct, then we play the sound with line 50. Remember, this is a sound object, not a sound stream, so when we play this sound, it gets played through the mixer, as all sounds pass through in order to play. Note that the mixer has no control over that specific sound, however, because it plays in its own separate channel.

Having said that, we can control certain aspects of the sounds with our mixer. For example, we can pause the playback of all sound, or stop it altogether, which leads us nicely onto our next section.

Stopping all of our sounds

We have done a great job of getting our code to produce a range of farmyard noises, but what can we do if we want to make these noises stop? Fortunately, we have a button to take care of this for us. On line 14 we have a dictionary called `stopButton`; unlike our soundboard buttons, it doesn't have a sound, just an image and a position element. That makes it special. Beneath all of the code used to handle our sound buttons, we have some code that only deals with the stop button: on line 28 we draw the stop button, just after we've drawn all of the sound ones, and on lines 53–55 we specifically check whether or not it is the stop button that has been clicked. Why do we give the stop button special treatment? The answer is that it is

unique, and for every button that doesn't do the same thing as all of the other buttons, we need to write custom code. Of course, we could use dictionaries and lists as we've done for our sound buttons, but that's far more complicated than is required for our purposes right now.

IT'S LOUD! Oh… it's quiet now…

So, we've loaded sounds, played them, and stopped them dead, but what if we just wanted to make the sounds a little quieter? This is simple enough to achieve. Each of our sound objects has a `set_volume()` function which can be passed a value between 0.0 and 1.0. 0.0 is mute, while 1.0 is full volume. If you pass a value larger than 1.0, it will become 1.0, and if you pass a value less than 0.0, it will become 0.0. To begin, we need to make a volume slider. On lines 30–36 we draw two rectangles. The first rectangle represents the range of 0.0 to 1.0, and the second rectangle is an indicator of the current volume. When we first start our soundboard, the volume (which is set on line 17 of our code) is set at 1.0, so our indicator should be all the way over on the right, but that doesn't do us much good if we want to turn things down. Just before we call `drawVolume()` on line 104, we call `checkVolume()` on line 103. Here we look at the current position of the mouse and whether or not the left mouse button is held down. If it is, our user is likely trying to drag our indicator to the level they want the sound to be at. So we work out where the mouse is on between 0.0 and 1.0 on our indicator and set the volume to the new level. Then, when our `drawVolume` function is called, the indicator will be drawn at the correct position. Now, when we next click a sound, it will be set to the level we've chosen with the `set_volume()` function on our sound object, which you can see on line 50.

And that's it. We've learned everything we need to know about making sounds play in our game. We've covered background audio, the sound mixer, streaming sound, and sound objects. We've also used lists and dictionaries to create and manipulate buttons, building on what we learned in the previous chapter. Now, we have a fully functioning soundboard which could form part of a game.

If you want a challenge, see if you can write code using what you've learned in the book so far to trigger the animal sounds using the keys 1–9 on your keyboard.

Sounds.py

Download
magpi.cc/1sgf4Rf

```
01. import pygame, sys, random
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05.
06. windowWidth = 600
07. windowHeight = 650
08.
09. pygame.init()
10. surface = pygame.display.set_mode((windowWidth, windowHeight))
11. pygame.display.set_caption('Soundboard')
12.
13. buttons = []
14. stopButton = { "image" : pygame.image.load(
    "assets/images/stop.png"), "position" : (275, 585)}
15.
16. mousePosition = None
17. volume = 1.0
18.
19. pygame.mixer.init()
20. pygame.mixer.music.load('assets/sounds/OGG/farm.ogg')
21. pygame.mixer.music.play(-1)
22.
23. def drawButtons():
24.
25.   for button in buttons:
26.     surface.blit(button["image"], button["position"])
27.
28.   surface.blit(stopButton["image"], stopButton['position'])
29.
30. def drawVolume():
31.
32.   pygame.draw.rect(surface, (229, 229, 229), (450, 610, 100, 5))
33.
34.   volumePosition = (100 / 100) * (volume * 100)
35.
36.   pygame.draw.rect(surface, (204, 204, 204), (
    450 + volumePosition, 600, 10, 25))
37.
38. def handleClick():
39.
40.   global mousePosition, volume
```

```
41.
42.   for button in buttons:
43.
44.     buttonSize = button['image'].get_rect().size
45.     buttonPosition = button['position']
46.
47.     if mousePosition[0] > buttonPosition[0] and
    mousePosition[0] < buttonPosition[0] + buttonSize[0]:
48.
49.       if mousePosition[1] > buttonPosition[1] and
    mousePosition[1] < buttonPosition[1] + buttonSize[1]:
50.         button['sound'].set_volume(volume)
51.         button['sound'].play()
52.
53.     if mousePosition[0] > stopButton['position']
    [0] and mousePosition[0] < stopButton['position'][0] +
    stopButton['image'].get_rect().size[0]:
54.       if mousePosition[1] > stopButton['position']
    [1] and mousePosition[1] < stopButton['position'][1] +
    stopButton['image'].get_rect().size[1]:
55.         pygame.mixer.stop()
56.
57. def checkVolume():
58.
59.   global mousePosition, volume
60.
61.   if pygame.mouse.get_pressed()[0] == True:

62.     if mousePosition[1] > 600 and mousePosition[1] < 625:
63.       if mousePosition[0] > 450 and mousePosition[0] < 550:
64.         volume = float((mousePosition[0] - 450)) / 100
65.
66. def quitGame():
67.   pygame.quit()
68.   sys.exit()
69.
70. # Create Buttons
71. buttons.append({ "image" : pygame.image.load(
    "assets/images/sheep.png"), "position" : (25, 25), "sound" :
    pygame.mixer.Sound('assets/sounds/OGG/sheep.ogg')})
73. buttons.append({ "image" : pygame.image.load(
    "assets/images/rooster.png"), "position" : (225, 25), "sound" :
    pygame.mixer.Sound('assets/sounds/OGG/rooster.ogg')})
74. buttons.append({ "image" : pygame.image.load(
    "assets/images/pig.png"), "position" : (425, 25), "sound" :
```

```
    pygame.mixer.Sound('assets/sounds/OGG/pig.ogg')})
75. buttons.append({ "image" : pygame.image.load(
    "assets/images/mouse.png"), "position" : (25, 225), "sound"
    : pygame.mixer.Sound('assets/sounds/OGG/mouse.ogg')})
76. buttons.append({ "image" : pygame.image.load(
    "assets/images/horse.png"), "position" : (225, 225), "sound"
    : pygame.mixer.Sound('assets/sounds/OGG/horse.ogg')})
77. buttons.append({ "image" : pygame.image.load(
    "assets/images/dog.png"), "position" : (425, 225), "sound" :
    pygame.mixer.Sound('assets/sounds/OGG/dog.ogg')})
78. buttons.append({ "image" : pygame.image.load(
    "assets/images/cow.png"), "position" : (25, 425), "sound" :
    pygame.mixer.Sound('assets/sounds/OGG/cow.ogg')})
79. buttons.append({ "image" : pygame.image.load("assets/images/
    chicken.png"), "position" : (225, 425), "sound" : pygame.
    mixer.Sound('assets/sounds/OGG/chicken.ogg')})
80. buttons.append({ "image" : pygame.image.load("assets/images/
    cat.png"), "position" : (425, 425), "sound" : pygame.mixer.
    Sound('assets/sounds/OGG/cat.ogg')})

81. # 'main' loop
82. while True:
83.
84.   surface.fill((255,255,255))
85.
86.   mousePosition = pygame.mouse.get_pos()
87.
88.   for event in GAME_EVENTS.get():
89.
90.     if event.type == pygame.KEYDOWN:
91.
92.       if event.key == pygame.K_ESCAPE:
93.         quitGame()
94.
95.     if event.type == GAME_GLOBALS.QUIT:
96.       quitGame()
97.
98.     if event.type == pygame.MOUSEBUTTONUP:
99.       handleClick()
100.
101.  drawButtons()
102.  checkVolume()
103.  drawVolume()
104.
105.  pygame.display.update()
```

The MagPi ESSENTIALS

[CHAPTER SIX]
PHYSICS & FORCES

In chapter six, we give our game objects mass and let gravity come into play.

In previous chapters, we've put together code that let us take control of elements in our program whenever we interact with them, be it by clicking, dragging or typing. The difficulty is, there's only so much we can do with these interactions; no matter what, everything we do will be determined by ourselves in some way, and that can get a little bit boring after a while. This being the case, in this chapter we're going to give certain elements of our program the ability to interact with things around them without us having to do anything: we're going to add gravity (or rather, motion that really closely resembles gravity) to some planets that we're going to make as part of a solar system simulator.

We must acknowledge a debt of gratitude to Daniel Shiffman for the inspiration behind this chapter. His book *The Nature of Code* explains the concepts found here and more in far greater detail. All of his code is written in Java (Processing), but you should be able to convert it to Python with a little bit of work.

Understanding gravity

You may be thinking that we have already covered the subject of gravity in chapter three. This is only partly the case. There, we added a force which we called gravity to certain objects to make them fall to the bottom of the window. However, that force was not particularly dynamic: no matter the object's size or velocity, it would simply add to the Y value of an object until it reached the bottom of the screen, which is not very interesting. For this new kind of gravity, we're going to be using something called vectors. A vector is a value which describes two things: direction and magnitude. With these, we can calculate the effect of one object with mass on the velocity (the speed and direction) of another object with mass. This program is bigger than anything we've made before, but we're not going to look at a great deal of it. Most of the things, such as drawing images or handling key presses, we've done before, so we're not going to revisit them; instead, we're going to focus on describing gravity and implementing that in our code. This is going to take some serious thinking, so if you don't understand everything the very first time, don't worry: almost everyone else who reads this will probably feel like they might need to go through it again.

⚠ CAUTION: MATHS AHEAD

So, what is this 'gravity' business, anyway?

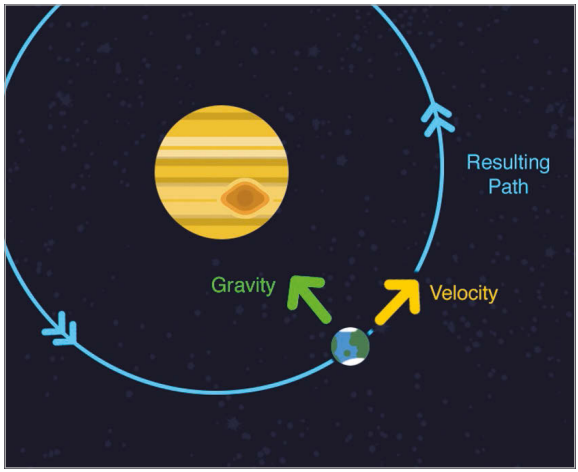
In the real world, gravity follows a rule called the inverse square law, which is this:

"The gravitational attraction between two point masses is directly proportional to the product of their masses and inversely proportional to the square of the separation distance. The force is always attractive and acts along the line joining them."

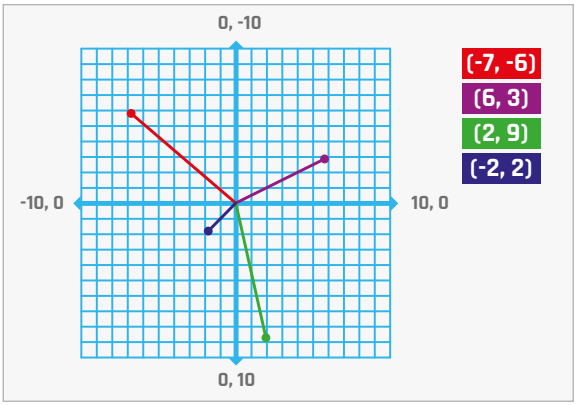
This seems like a very complicated concept, but what does it mean? It's actually really simple: it means the force acting on something reduces as the distance increases. Thus, however strong the pull of gravity on something is – like, for example, the Earth pulling on a football 1 foot in the air – if we were to move the same object so that it was 3 feet away from the gravity source, the force will be 1/9th as strong, that is 1/3 squared or 1 / distance^2. The same law applies to more than just gravity – it affects light and sound too – but that's not relevant here. What is important is you should now be begining to get a feeling for how gravity should work: further away means less force, closer means more force. Equally important is the final sentence of that statement:

"The force is always attractive and acts along the line joining [the point masses]."

Gravity always pulls, and never repels. It always pulls in the direction of the objects it is pulling. It is because of this truth that we're going to use vectors to simulate gravity. Using vectors, we can calculate the direction of each object in relation to another and adjust it to the force of gravitational attraction accordingly. The result is that gravity happens.



Below An illustration demonstrating the gravitational attraction of two bodies, one orbiting the other



Above A diagram showing vectors on a grid

V is for vector

So now we've got an understanding of how gravity works, it's time to take a look at what a vector is. You can think of a vector like an arrow: it has two values, an X and a Y, and together these point in a direction. For example, if we were to draw a line from (0,0) along a vector of (8, 4) on a grid, it would point down and to the right; for every unit travelled along the X axis (pixels, centimetres, inches, fathoms, the unit type doesn't matter), 0.5 units would be travelled along the Y axis. If we were to draw another line from (0,0) along a vector of (-1, -2), the line would travel to the left and up; for each unit travelled along the X axis, two would be traversed along the Y axis.

So with vectors we can describe direction, but we can also express something called magnitude. The magnitude of the vector is the length of the line drawn between (0,0) and the vector on a grid, but we can also think of the magnitude as an amount or size of something; for example, we could use it as speed.

> "The magnitude of the vector is the length of the line drawn"

When we use vectors to describe direction, it often helps to normalise them. This means we take a vector, such as (1, 3), and turn each value into a value somewhere between -1 and 1 by dividing it by the magnitude. For instance, the vector (1, 3) would be normalised to (0.316, 0.948), while (-8, 2.4) would normalise to (-0.957, 0.287). Normalising our values in this way makes it much easier to affect things with force and direction. By having a value between -1 and 1, we have only an indication of direction. When we have that, we're free to adjust it by any value to suit our needs; for instance, we could multiply the values by a speed value to give something motion.

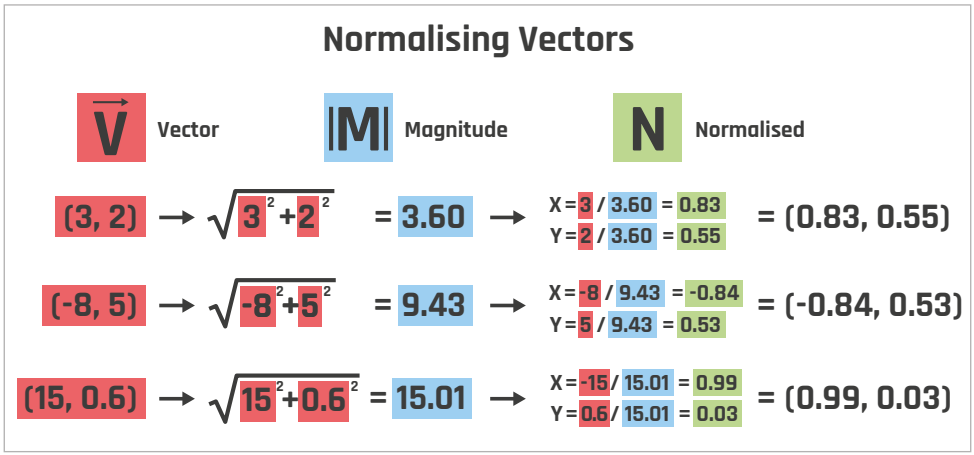
A speedy overview

To review the material we have just covered: gravity always attracts in the direction of something with a mass; vectors describe a direction and a magnitude which is an amount of something, such as speed; and vectors can be shrunk down to a value between -1 and 1 to describe only a direction, through a process called normalisation. Now that we have revised and understood these key concepts, it is time to start looking at some code with Pygame.

As we said earlier, we're going to skip over explaining a lot of the code for this tutorial – it is all material we have looked at before – but for the sake of clarity we'll do a quick pass over the functions, what they do, and the order they're called in. With the knowledge we've gained above, we're going to construct a solar system simulator that moves planets around with gravity, based on their mass and velocity.

On lines 1–30 of **simulator.py** we have all of the variables we need to run our program. The `import` statements at the top of our script are almost identical to our previous programs, with one exception: `import solarsystem` on line 5. This is not a module like the other `import` statements, but rather a custom script written for this tutorial, and you can grab it from GitHub. Just drop it in the same folder as simulator.py; it simply creates new planets for our simulator and doesn't need to be in the main simulator.py code, as our games are going to start to get messy if everything is in one script!

Below The maths of calculating the magnitude of a vector and normalising it



Above Our Solar System Simulator on its first run

Lines 31–54 contain the functions `drawUI()`, `drawPlanets()` and `drawCurrentBody()`. These are responsible for drawing the elements of our program to our window. All of these are called once every time the main loop runs, in the order `drawUI()`, `drawPlanets()`, and then `drawCurrentBody()`. The `currentBody` function is responsible for drawing the planet that the user is currently dragging around the window, before letting it go to affect other planets with its gravity.

Lines 56–82 involve the `calculateMovement()` function. It is here that we make all of the gravity magic happen. It gets called in the main loop, just before `drawPlanets()`. This is the clever bit of our program and we'll work through every single line in a moment.

Lines 84–106 handle the mouse and system events. When our player clicks somewhere in our window, `handleMouseDown()` is run and checks whether or not our user clicked in one of the planet tabs at the bottom of our window with `checkUIForClick()`. If they have, `checkUIForClick()` will return the name of that planet and it will be created with `solarsystem.makeNewPlanet()`, the only function that we imported with `import solarsystem` at the start of our script.

Finally we have lines 109–165, our familiar 'main' loop. Just like in our previous programs, it is from here that we call functions to handle user interactions and update our surface. The function calls on lines 146–157 are where we update and draw our planets.

The movement of the Spheres

So, let's get to work. If you fire up the simulator.py script, you'll see our Solar System Simulator. After four seconds, the logo will disappear and you'll then be able to drag one of the eight planets at the bottom to somewhere on the screen. Each planet has characteristics which loosely reflect those of its real-world counterpart. Jupiter has the

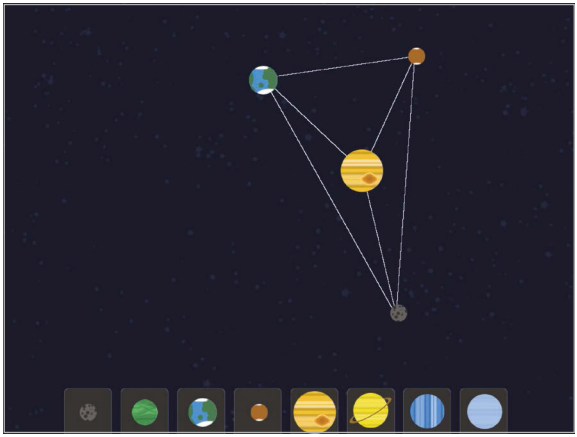
greatest mass, Mercury has the least, Venus is only slightly smaller than Earth, Mars is a third of Earth's size and so on. By clicking on a planet, we create a new planet which is stored in the `currentBody` variable. The latter lets us create and place a planet without affecting any other planets on the screen. It's only when we let go of the planet that gravity is allowed to take effect on the new body in the solar system. All of this neat gravitational magic happens on lines 56–82.

No matter how far apart two objects are in the universe, they still have a gravitational pull on one another, even if it is infinitesimally small. If you were to place two dice a metre apart in space and leave them to their own devices, eventually the pull of the dice on one another would bring them both together without any help from another force. This effect is replicated in our simulator. Each planet in our solar system has a gravitational effect on every other body in our system. To do this, we create a `for` loop that works through every planet in our `celestialBodies` list on line 58.

For each planet we have, we want to calculate its effect on every other planet in our system, so the first thing our `for` loop on line 58 does is create another loop to work through the rest of the planets. We don't want to calculate the effect of a planet on itself, so before we start our calculations we need to check that `otherPlanet` is not the same as the planet we're using for the gravitational pull calculations. Once we have a valid planet to affect (we will refer to it here as otherPlanet), we can start working with numbers and figuring out some vectors.

The first thing we need to find is the vector between the planet and otherPlanet. We do this on line 64 with the variable `direction`. The variable is named 'direction' because it points from the coordinates of our planet to the coordinates of the otherPlanet that we're trying to affect. Once we have the direction, we can work out the magnitude (in this case, the distance) between the two planets.

To help us work out the magnitude of our direction vector, we can use Python's built-in maths library, specifically the `hypot` method which we use on line 65. If you're curious about the actual formula for figuring out the magnitude, it's the square root of the X squared and Y squared coordinates added to each other. Once we have our magnitude, we can use it to normalise our direction vector. Normalising our vector means we'll have vector X and Y values that are proportional to one another but fall between -1 and 1. This is useful for us, because that



Above The lines of attraction drawn between planets

lets us multiply our vector by any value we wish to affect our force. To normalise our vector, all we have to do is divide our direction vector by the magnitude, and we do that on line 66 with the variable **nDirection**.

We have almost everything we need to start applying gravity, but before we do, we should limit magnitude. Strange things happen when forces are very big or very small, even in simulators, so we set a maximum for the number that magnitude can be on lines 68–72.

We now have all we need to apply gravity to our planet. However, at this point, we'd be applying an arbitrary value that had nothing to do with the properties of our planets. What we want to do now is take into consideration the mass of our objects, because gravity only affects things with mass.

On line 74 we have the **strength** variable. Here, we calculate how much force we need to apply to each planet to generate gravity. First, we multiply the planet's mass by the otherPlanet's mass and multiply that by our **gravity** variable on line 29. The gravity value is arbitrary and we can tweak it to generate stronger or weaker gravitational effects: remember, we're creating the illusion of gravity, not actually modelling the universe. Next, we divide that value by **magnitude** squared: this enables our objects to accelerate as they approach one another. Finally, we divide all of that by the mass of the planet we're affecting, otherPlanet. This lets our objects move slower if they are dense, and faster if they are less dense. By making it harder to move the big planets, we avoid small planets towing much larger ones around.

We now have the values we need to apply gravity to our planets. On line 75, we create a new vector. By multiplying our normalised direction vector (**nDirection**) by the **strength** value, we now have a vector with both direction and magnitude determined by the gravitational attraction of our objects. On lines 77 and 78 we apply this

new vector to the velocities of our otherPlanet; the next time our planet is drawn, its position will have been adjusted by gravity.

The last section of `calculateMovement()` on lines 80 to 82 doesn't have anything to do with moving the planets: it simply draws a line between our planet and every otherPlanet that it's having an effect on. It's the line of attraction we looked at earlier, and it illustrates the directions that gravity is pulling our planets in. You can toggle this on and off with the 'A' key.

Rounding up

We have covered rather a lot of material in this chapter. We have learned all about vectors and how we can use them to determine both speed and direction, rather like velocity. We have also learned how to normalise values so they can be made to do our bidding through multiplication. We have learned about how gravity works in the real world, and how we can emulate that in our simulated world. We also have some pretty neat code for handling our mouse and keyboard events. It may have been complicated, but hopefully you are getting a sense of what you can do with Pygame.

Simulator.py

Download
magpi.cc/1jQkp8m

```
01. import pygame, sys, random, math
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05. import solarsystem
06.
07. windowWidth = 1024
08. windowHeight = 768
09.
10. pygame.init()
11. surface = pygame.display.set_mode((
    windowWidth, windowHeight), pygame.FULLSCREEN)
12.
13. pygame.display.set_caption('Solar System Simulator')
14.
15. previousMousePosition = [0,0]
16. mousePosition = None
17. mouseDown = False
```

```
18.
19.  background = pygame.image.load("assets/background.jpg")
20.  logo = pygame.image.load("assets/logo.png")
21.  UITab = pygame.image.load("assets/tabs.png")
22.  UICoordinates = [{"name" : "mercury", "coordinates" :
     (132,687)}, {"name" : "venus", "coordinates" : (229,687)},
     {"name" : "earth", "coordinates" : (326,687)},
     {"name" : "mars", "coordinates" : (423,687)},
     {"name" : "jupiter", "coordinates" : (520,687)},
     {"name" : "saturn", "coordinates" : (617,687)},
     {"name" : "neptune", "coordinates" : (713,687)},
     {"name" : "uranus", "coordinates" : (810,687)}]
23.
24.  celestialBodies = []
25.  currentBody = None
26.
27.  drawAttractions = True
28.
29.  gravity = 10.0
30.
31.  def drawUI():
32.        surface.blit(UITab, (131,687))
33.        surface.blit(solarsystem.images["mercury"], (158,714))
34.        surface.blit(solarsystem.images["venus"], (247,706))
35.        surface.blit(solarsystem.images["earth"], (344,704))
36.        surface.blit(solarsystem.images["mars"], (451,714))
37.        surface.blit(solarsystem.images["jupiter"], (524,692))
38.        surface.blit(solarsystem.images["saturn"], (620,695))
39.        surface.blit(solarsystem.images["neptune"], (724,697))
40.        surface.blit(solarsystem.images["uranus"], (822,697))
41.
42.  def drawPlanets():
43.
44.        for planet in celestialBodies:
45.              planet["position"][0] += planet["velocity"][0]
46.              planet["position"][1] += planet["velocity"][1]
47.              surface.blit(solarsystem.images[planet["name"]],
     (planet["position"][0] - planet["radius"],
     planet["position"][1] - planet["radius"]))
48.
49.  def drawCurrentBody():
50.
51.        currentBody["position"][0] = mousePosition[0]
52.        currentBody["position"][1] = mousePosition[1]
53.
54.        surface.blit(solarsystem.images[currentBody["name"]],
```

```
    (currentBody["position"][0] - currentBody["radius"],
    currentBody["position"][1] - currentBody["radius"]))
55.
56. def calculateMovement():
57.
58.     for planet in celestialBodies:
59.
60.         for otherPlanet in celestialBodies:
61.
62.             if otherPlanet is not planet:
63.
64.                 direction = (
    otherPlanet["position"][0] - planet["position"][0],
    otherPlanet["position"][1] - planet["position"][1])
    # The difference in the X, Y coordinates of the objects
65.                 magnitude = math.hypot(
    otherPlanet["position"][0] - planet["position"][0],
    otherPlanet["position"][1] - planet["position"][1])
    # The distance between the two objects
66.                 nDirection = (
    direction[0] / magnitude, direction[1] / magnitude)
    # Normalised vector pointing in the direction of force
67.
68.                 # We need to limit the gravity
                    if magnitude < 5:
69.                     magnitude = 5
70.                 elif magnitude > 30:
71.                     magnitude = 30
72.
73.                 strength = ((
74. gravity * planet["mass"] * otherPlanet["mass"]) / (
    magnitude * magnitude)) / otherPlanet["mass"]
    # How strong should the attraction be?

75.                 appliedForce = (
76. nDirection[0] * strength, nDirection[1] * strength)

77.                 otherPlanet["velocity"][0] -=
    appliedForce[0]
79.                 otherPlanet["velocity"][1] -=
    appliedForce[1]

80.                 if drawAttractions is True:
81.                     pygame.draw.line(
82. surface, (255,255,255),
    (planet["position"][0],planet["position"][1]),(
```

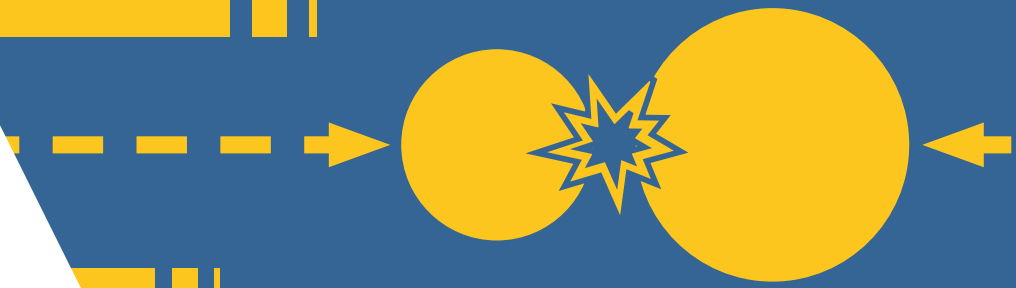
```
    otherPlanet["position"][0],otherPlanet["position"][1]), 1)
83.
84. def checkUIForClick(coordinates):
85.
86.       for tab in UICoordinates:
87.             tabX = tab["coordinates"][0]
88.
89.             if coordinates[0] > tabX and coordinates[0] < tabX
    + 82:
90.                   return tab["name"]
91.
92.       return False
93.
94. def handleMouseDown():
95.       global mousePosition, currentBody
96.
97.       if(mousePosition[1] >= 687):
98.             newPlanet = checkUIForClick(mousePosition)
99.
100.            if newPlanet is not False:
101.                  currentBody = solarsystemmakeNewPlanet(
    newPlanet)
102.
103.
104. def quitGame():
105.   pygame.quit()
106.   sys.exit()
107.
108. # 'main' loop
109. while True:
110.
111.   mousePosition = pygame.mouse.get_pos()
      surface.blit(background, (0,0))
112.
113.   # Handle user and system events
114.   for event in GAME_EVENTS.get():
115.
116.     if event.type == pygame.KEYDOWN:
117.
118.       if event.key == pygame.K_ESCAPE:
119.         quitGame()
120.
121.     if event.type == pygame.KEYUP:
122.
123.       if event.key == pygame.K_r:
124.             celestialBodies = []
```

```
125.      if event.key == pygame.K_a:
126.           if drawAttractions is True:
127.                drawAttractions = False
128.           elif drawAttractions is False:
129.                drawAttractions = True
130.
131.     if event.type == pygame.MOUSEBUTTONDOWN:
132.       mouseDown = True
133.       handleMouseDown()
134.
135.     if event.type == pygame.MOUSEBUTTONUP:
136.        mouseDown = False
137.
138.     if event.type == GAME_GLOBALS.QUIT:
139.       quitGame()
140.
141.   # Draw the UI, update the movement of the planets,
142.   # draw the planets in their new positions.
      drawUI()
143.   calculateMovement()
144.   drawPlanets()
145.
146.   # If our user has created a new planet,
147.   # draw it where the mouse is
      if currentBody is not None:
148.      drawCurrentBody()
149.
150.      # If our user has released the mouse, add the new
151.      # planet to the celestialBodies list
          if mouseDown is False:
152.           currentBody["velocity"][0] = (
    mousePosition[0] - previousMousePosition[0]) / 4
153.           currentBody["velocity"][1] = (
154. mousePosition[1] - previousMousePosition[1]) / 4
               celestialBodies.append(currentBody)
155.           currentBody = None
156.
157.      # Draw the logo for the first four seconds
158.   if GAME_TIME.get_ticks() < 4000:
         surface.blit(logo, (108,77))
159.
160.   # Store the previous mouse coordinates to create a
161.   # vector when we release a new planet
162.   previousMousePosition = mousePosition

163.   pygame.display.update()
```

The MagPi ESSENTIALS

[CHAPTER SEVEN]
PHYSICS & COLLISIONS

In chapter seven, we create circles which can bounce off one another if they collide.



What happens when an unstoppable force meets an immovable object?

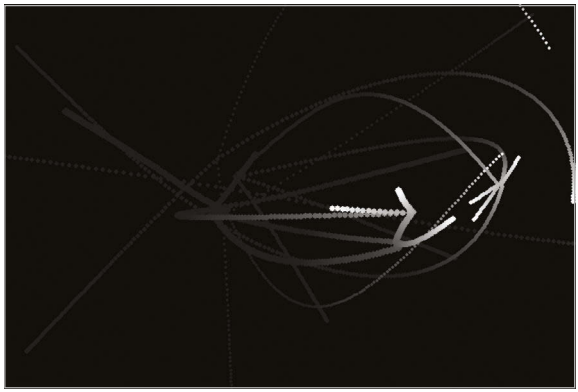
In our last chapter, we simulated a sizeable amount of a solar system. Using vectors and maths, we created a gravitational attraction between objects with mass to simulate their movement in space. Small objects would orbit larger ones, large objects would move very little when attracted to smaller objects and vice versa, and all was well in the simulated world. That said, one thing might have seemed a little odd: when two objects collide in the real world, they bounce off one another (or implode), but in our simulation they just slipped by one another as if they were ghosts. This time, howver, we're going to write code that lets our objects collide and bounce off each other.

So, what are we making?

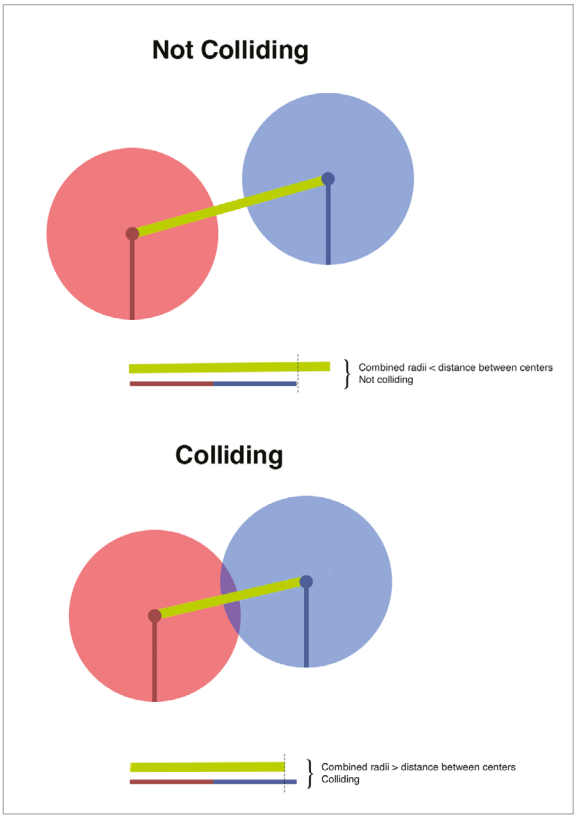
Unlike last time, we aren't going to be using planets and the solar system to prettify the effect – we're going to use basic circles for our program. Using circles makes it easier for us to use maths to calculate collisions, and we can change the properties of the circles to reflect the qualities they represent: for example, more mass or a bigger radius. That said, although we aren't using images of the solar system in this program, we can still think of the particles we'll be colliding in terms of a solar system.

The smallest of our collidable objects will be like meteors: they move really fast, but require less energy to do so. A medium-size object would behave much as a planet might; they move at a moderate speed and have more kinetic energy behind them. If they bump into a smaller object, they will adjust course, but not by much, whereas the smaller body will fly off!

We're going to use the code from the last tutorial as a springboard for this one. All of our objects will have a mass and will attract every other object



Below Simulating object collisions



Above
A diagram of all of the aspects needed to bounce one circle off another

gravitationally using the same `calculateMovement()` method as before.

Let's take a quick walk through our code now. Just like our previous bits of code, the top of **collisions.py** (between lines 1–24) imports the modules we'll need for our code and declares the variables that we'll be using throughout the tutorial. Obviously, these variables are very similar to the variables we used for our solar system simulator, but there's one little difference: instead of storing all of our planets in a list called `celestialBodies`, this time we're storing all of our objects in the `collidables` list.

Lines 26–84 will also seem familiar. `drawCollidables` is the same function as our `drawPlanets()` function, our `drawCurrentBody()` hasn't changed at all, and the same can be said for our `calculateMovement()` function which handles the gravity of all of our objects.

The `handleCollisions()` function on lines 86–135 is where we'll spend our time this tutorial. Here, we check for colliding objects and adjust their trajectories accordingly.

Lines 137–202 contain the logic for our keyboard and mouse interactions, as well as our main loop. Just as before, clicking in our window will create a new particle which will only affect the movement of other particles once the mouse has been released. If the mouse was moving when it was released, the particle will inherit the velocity of the mouse pointer.

What do we need to know to simulate a collision?

We need to know a couple of things before we can simulate a collision. First, we need to know which two objects, if any, are colliding. Once we know which two objects are colliding, we need to figure out how fast they're going, the angle of incidence (which we'll look at in a little while), and the mass of each of the objects.

So, how do we know which two objects are colliding? This pretty straightforward when you use circles. Circles are regular shapes: each point along the circumference is the same distance from the centre as every other point; this measurement from the edge to the centre is the radius. By measuring the distance between the centres of two objects, we can check whether or not the outlines of the objects are intersecting. If the distance between the centres of two circles is less than the radius of each circle added to the other, we can conclude that they must be colliding.

On lines 88–96, we create two `for` loops that let us work through every possible collidable object in our simulation. Inside these loops, we measure the distance between the centres of every object in our `collidables` list. We do this on line 102 with our `distance` variable, using Python's maths module. If the distance between the two centres of our objects is more than the combined length of the radius of each circle, our objects are not colliding, and we then continue on measuring the distance to other circles. However, if the distance is less than the sum of the radii, then our objects are colliding and we can start figuring out what to do with them.

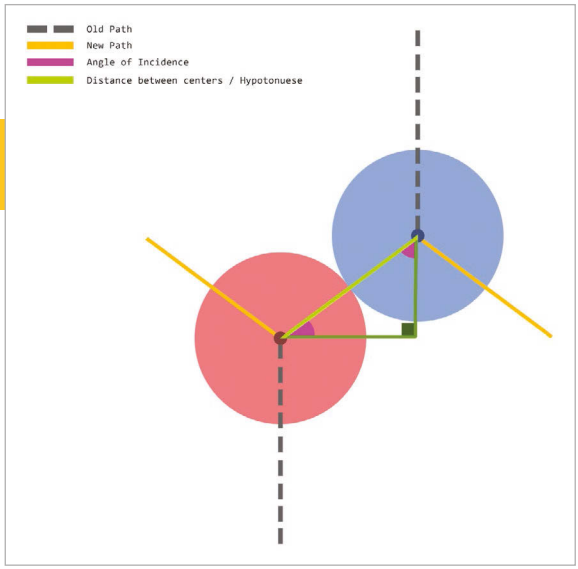
[THANKS]

A hat tip for this chapter goes out to Steve and Jeff Fulton. They put a huge amount of effort into dissecting old Flash-based physics code into its core parts and putting it back together for their book *HTML5 Canvas*, which made this chapter possible.

⚠ WARNING! PYTHAGORAS' THEOREM AHEAD

The angle of incidence

How do we create a convincing bounce effect? In the past, whenever we've wanted to restrict or adjust movement, we've been doing so with squares or rectangles – shapes with flat sides. Going too much to the right? Okay, we'll send it back to the left. Going too far down? Okay, we'll send it back up. Now, we're dealing with circles. Instead of having eight directions to choose from, we now have 360. If our circles hit each other square on, then all we have to do is send them back along



Above A right-angled triangle is used to figure out the angle of incidence

the path they came, but these are circles with gravity; hitting another circle square on along the X or Y axis is not going to happen very often. If a collision happens a little to the left or right of the centre of the X or Y axis, we need to send our objects on two new paths, but how do we know which direction to send each object? For this, we need to use the angle of incidence: this is the angle at which an object is travelling as it collides with another object. If we know the angle at which two things collide, we can figure out along which angle we can send them on their way onward: this is the angle of reflection, which is the reverse of the angle of incidence.

This is not as complicated as it sounds. Imagine a ball hitting a vertical wall at an angle 45, so its vector is (1, 1), travelling to the right and down in equal measure. After the ball hits the wall, the rate at which the it falls to the ground is unchanged, but the direction it's travelling is reversed along its X axis; our ball is still travelling at 45 degrees, but now it's travelling away from the wall, at -45 degrees or with a vector of (-1, 1). We've coded this previously, when we first started bouncing objects around a window, but we weren't using the same terms we are here. Now we know why we made objects move the way we did.

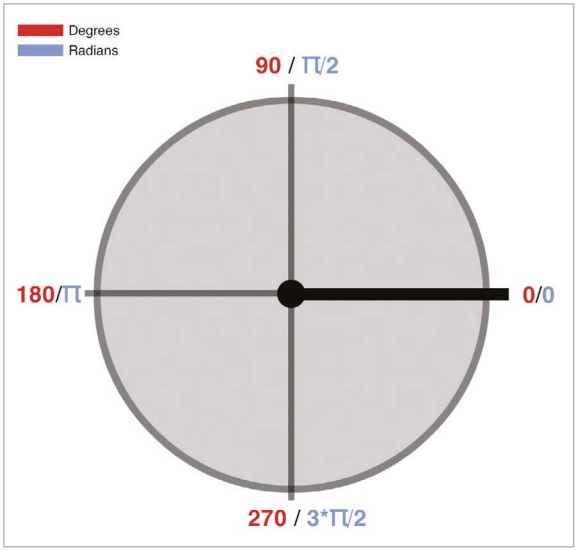
On line 107, we calculate the angle of incidence between the centre of the two circles colliding with `math.atan2`, which basically works out the hypotenuse of an imaginary right-angled triangle drawn using the two centre points of the circles. If you were to print out the value of the direction variable, you might expect it to read somewhere between 0 and 360 because an angle is measured in degrees. In fact, you'll get a value between 1 and 2π (pi * 2): our angle has been measured in radians. This may seem counter-intuitive, but

to a computer (and mathematicians) it makes perfect sense. If you want to see the degree value, you can simply do 'radians * (180/pi)', but we are going to stick with radians because it keeps our code tidy.

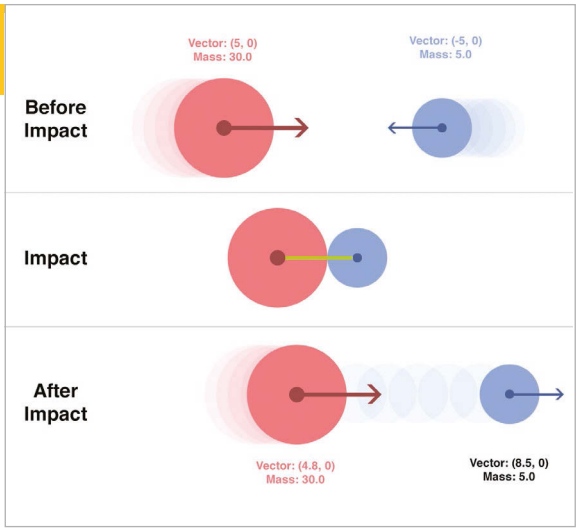
Bounce!

Now we've got the angle of incidence, we can calculate which way to send our colliding objects, but first we need to obtain a couple of other values to make everything work. Next, we need to work out the speed at which our circles are moving. You may wonder why, if we have vectors, we need a separate speed value. It is indeed true that we use vectors to affect the speed and direction of our objects, but that's part of the problem: our vector is a measure of both speed and direction. As it is, we can't use the vectors to find out how many pixels our objects travel per frame; we need to separate the speed from the direction so we can perform some maths specific to each value. Fortunately, we can use maths to figure out the speed of our objects – which we do on lines 110–111, one variable for each object in the collision – and the direction each object is moving in radians, on lines 114–115.

Now we have the speed and direction of each circle, we can adjust them separately to create the bouncing effect. First, we want to reverse the direction in which the objects are travelling; on lines 118–122, we create a couple of variables to calculate new velocities. Here we've recombined the speed and direction variables of each object to create new speeds for the X and Y values of our circles. When used together, we have our new vector. But these ones will point our objects in the opposite direction of the angle of incidence – the angle of reflection. We've got the basics of our bounce.



Below A diagram displaying angles on a circle and their equivalent values in radians



Above An illustration of the effect of mass on vectors in a collision

Motion

Energy cannot be destroyed, only converted from one form to another. Motion is a form of energy and when two objects collide, an energy transfer happens between them. Of the two objects colliding, the faster object will transfer energy into the slower object, speeding the slower object up and slowing itself down. The two objects will move off in different directions and at different speeds than they were travelling before the collision, but the net energy of motion – the total amount of energy moving the objects – will remain exactly the same; it's just in different quantities in different objects now.

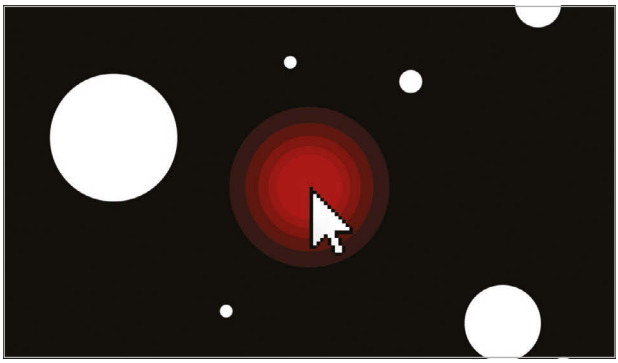
On lines 125 and 126, we take into consideration the mass of each of the objects colliding, as well as the speed. The result is that bigger objects will take more energy to change direction than smaller ones. With this in place, we won't have large objects being sent off at high velocities by much smaller, faster-moving objects, just like in the real world. Remember, this is all about simulating physics: if we wanted to calculate interactions between multiple objects scrupulously accurately and as if they existed in the real world, we'd need a computer much more powerful than our Raspberry Pi.

Now we have the new vectors for our colliding objects, all we have to do is apply them to our objects. We're only going to apply the X values we've calculated to each object. If we applied both the adjusted X and Y values to each object, they would bounce and follow the path they came along. That would be like throwing a ball and having it bounce straight back into your hand: it would be unnatural. By only applying the X value to each of our colliding objects, we can create a convincing, bouncing, deflecting effect, and we do that on lines 129 and 130.

And that's it: we can simply repeat this for every possible collidable object in our simulator.

[QUICK TIP]

We haven't used the planet graphics or much of the user interaction code that we wrote for the solar system, but, with a little work, you should be able to drop the `handleCollisions()` function into last chapter's code and make your planets bounce. Consider it a challenge!



Left Use the mouse to create a new moving object

Collisions.py

Download
magpi.cc/1jQkKYG

```
01. import pygame, sys, random, math
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05.
06. windowWidth = 1024
07. windowHeight = 768
08.
09. pygame.init()
10. surface = pygame.display.set_mode((
    windowWidth, windowHeight))
11.
12. pygame.display.set_caption('Collisions')
13.
14. previousMousePosition = [0,0]
15. mousePosition = None
16. mouseDown = False
17.
18. collidables = []
19. currentObject = None
20. expanding = True
21.
22. drawAttractions = False
23.
24. gravity = 1.0
25.
26. def drawCollidables():
27.
```

```
28.       for anObject in collidables:
29.             anObject[“position”][0] += anObject[“velocity”][0]
30.             anObject[“position”][1] += anObject[“velocity”][1]
31.
32.             pygame.draw.circle(surface, (255,255,255),(
    int(anObject[“position”][0]), int(
    anObject[“position”][1])), int(anObject[“radius”]), 0)
33.
34. def drawCurrentObject():
35.
36.       global expanding, currentObject
37.
38.       currentObject[“position”][0] = mousePosition[0]
39.       currentObject[“position”][1] = mousePosition[1]
40.
41.       if expanding is True and currentObject[
    “radius”] < 30:
42.             currentObject[“radius”] += 0.2
43.
44.             if currentObject[“radius”] >= 30:
45.                   expanding = False
46.                   currentObject[“radius”] = 9.9
47.
48.       elif expanding is False and currentObject[“radius”] > 1:
49.             currentObject[“radius”] -= 0.2
50.
51.             if currentObject[“radius”] <= 1:
52.                   expanding = True
53.                   currentObject[“radius”] = 1.1
54.
55.       currentObject[“mass”] = currentObject[“radius”]
56.
57.       pygame.draw.circle(surface,(255,0,0),(int(
    currentObject[“position”][0]), int(currentObject[
    “position”][1])), int(currentObject[“radius”]), 0)

58. def calculateMovement():
59.
60.       for anObject in collidables:
61.
62.             for theOtherObject in collidables:
63.
64.                   if anObject is not theOtherObject:
65.
66.                         direction = (theOtherObject[
67. “position”][0] - anObject[“position”][0], theOtherObject[
```

```
    "position"][1] -anObject["position"][1])


68. magnitude = math.hypot(
    theOtherObject["position"][0] - anObject["position"][0],
    theOtherObject["position"][1] - anObject["position"][1])
69.                         nDirection = (
    direction[0] / magnitude, direction[1] / magnitude)
70.
71.                         if magnitude < 5:
72.                             magnitude = 5
73.                         elif magnitude > 15:
74.                             magnitude = 15
75.
76.                         strength = ((
    gravity * anObject["mass"] * theOtherObject["mass"]) / (
    magnitude * magnitude)) / theOtherObject["mass"]
77.
78.                         appliedForce = (
    nDirection[0] * strength, nDirection[1] * strength)
79.
80.                         theOtherObject["velocity"][0] -= appliedForce[0]
81.                         theOtherObject["velocity"][1] -= appliedForce[1]
82.
83.                         if drawAttractions is True:
84.                             pygame.draw.line(
    surface, (255,255,255), (anObject["position"][0],anObject[
    "position"[1]), (theOtherObject["position"][0],theOtherObject["position"][1]), 1)
85.
86. def handleCollisions():
87.
88.       h = 0
89.
90.       while h < len(collidables):
91.
92.             i = 0
93.
94.             anObject = collidables[h]
95.
96.             while i < len(collidables):
97.
98.                   otherObject = collidables[i]
99.
100.                  if anObject != otherObject:
101.
102.                        distance = math.hypot(otherObject["position"][0] -
```

```
        anObject["position"][0], otherObject[
        "position"][1] - anObject["position"][1])
103.
104.                       if distance < otherObject[
        "radius"] + anObject["radius"]:
105.
106.                           # First we get the angle of the
                               # collision between two objects
107.                           collisionAngle = math.
        atan2(anObject["position"][1] - otherObject["position"][1],
        anObject["position"][0] - otherObject["position"][0])
108.
109.                           # Then we need to calculate the
                               # speed of each object
110.                           anObjectSpeed = math.
        sqrt(anObject["velocity"][0] * anObject["velocity"][0] +
        anObject["velocity"][1] * anObject["velocity"][1])
111.                           theOtherObjectSpeed = math.
        sqrt(otherObject["velocity"][0] * otherObject["velocity"][0]
        + otherObject["velocity"][1] * otherObject["velocity"][1])
112.
113.                           # Now, we work out the direction
                               # of the objects in radians
114.                           anObjectDirection = math.
        atan2(anObject["velocity"][1], anObject["velocity"][0])
115.                           theOtherObjectDirection = math.
        atan2(otherObject["velocity"][1], otherObject["velocity"]
        [0])
116.
117.                           # Now calculate new X/Y values
                               # of each object for collision
118.                           anObjectsNewVelocityX =
        anObjectSpeed * math.cos(anObjectDirection - collisionAngle)
119.                           anObjectsNewVelocityY =
        anObjectSpeed * math.sin(anObjectDirection - collisionAngle)
120.
121.                           otherObjectsNewVelocityX =
        theOtherObjectSpeed * math.cos(theOtherObjectDirection -
        collisionAngle)
122.                           otherObjectsNewVelocityY =
        theOtherObjectSpeed * math.sin(theOtherObjectDirection -
        collisionAngle)
123.
124.                           # We adjust the velocity based
                               # on the mass of the objects
```

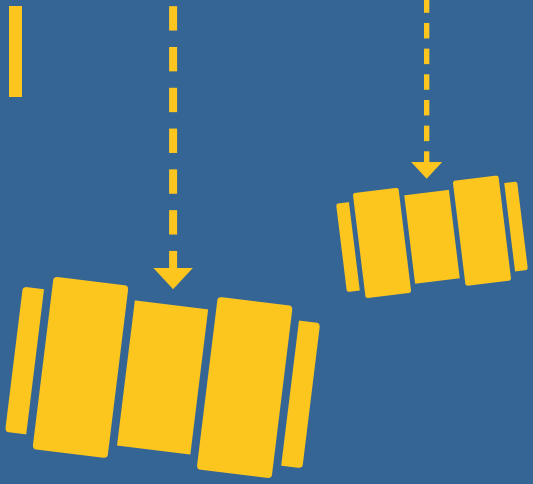
```
                        anObjectsFinalVelocityX
= ((anObject["mass"] - otherObject["mass"]) *
anObjectsNewVelocityX + (otherObject["mass"] +
otherObject["mass"]) * otherObjectsNewVelocityX)/
(anObject["mass"] + otherObject["mass"])
                        otherObjectsFinalVelocityX =
((anObject["mass"] + anObject["mass"]) * anObjectsNewVelocityX
+ (otherObject["mass"] - anObject["mass"]) *
otherObjectsNewVelocityX)/(anObject["mass"] + otherObject["mass"])

                        # Now we set those values
                        anObject[
"velocity"][0] = anObjectsFinalVelocityX
                        otherObject[
"velocity"][0] = otherObjectsFinalVelocityX


                i += 1

           h += 1

def handleMouseDown():
     global currentObject

     currentObject = {
          "radius" : 3,
          "mass" : 3,
          "velocity" : [0,0],
          "position" : [0,0]
     }

def quitGame():
  pygame.quit()
  sys.exit()

# 'main' loop
while True:

  surface.fill((0,0,0))
  mousePosition = pygame.mouse.get_pos()

  # Handle user and system events
  for event in GAME_EVENTS.get():

    if event.type == pygame.KEYDOWN:

```

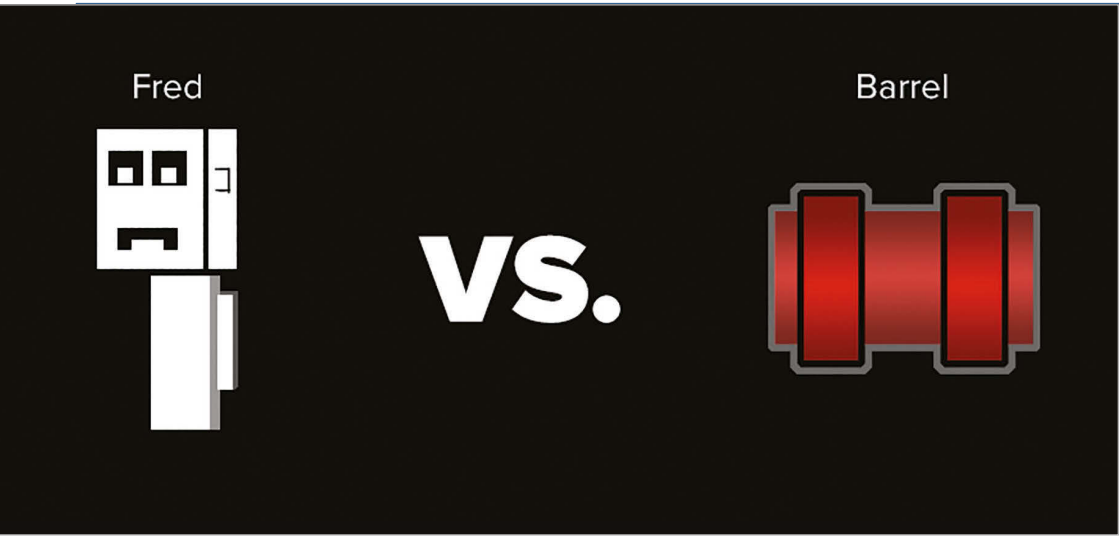
```
162.          if event.key == pygame.K_ESCAPE:
163.            quitGame()
164.
165.        if event.type == pygame.KEYUP:
166.
167.          if event.key == pygame.K_r:
168.               collidables = []
169.          if event.key == pygame.K_a:
170.               if drawAttractions is True:
171.                    drawAttractions = False
172.               elif drawAttractions is False:
173.                    drawAttractions = True
174.
175.        if event.type == pygame.MOUSEBUTTONDOWN:
176.          mouseDown = True
177.          handleMouseDown()
178.
179.        if event.type == pygame.MOUSEBUTTONUP:
180.          mouseDown = False
181.
182.        if event.type == GAME_GLOBALS.QUIT:
183.          quitGame()
184.
185.      calculateMovement()
186.      handleCollisions()
187.      drawCollidables()
188.
189.      if currentObject is not None:
190.          drawCurrentObject()
191.
192.          # If our user has released mouse, add the new
             # new anObject to the collidables list and
             # let gravity do its thing
193.          if mouseDown is False:
194.               currentObject["velocity"][0] = (
mousePosition[0] - previousMousePosition[0]) / 4
195.               currentObject["velocity"][1] = (
mousePosition[1] - previousMousePosition[1]) / 4
196.               collidables.append(currentObject)
197.               currentObject = None
198.
199.      # Store the previous mouse coordinates to create a
         # vector when we release a new anObject
200.      previousMousePosition = mousePosition
201.
202.      pygame.display.update()
```

The MagPi ESSENTIALS

[CHAPTER EIGHT] CLASSES

In chapter eight, we make a simple barrel-dodging game using classes.





Going further with game design

We are now over three quarters of the way through this book, which you can look at in two ways: on one hand, we are drawing close to the end, but on the other, we still have several opportunities to learn and make something amazing. We've put together so much already at this point: we've learned all about how Pygame draws shapes and images, how we can manipulate sounds and control events with our keyboards and mouse, we've made buttons and start screens and life bars and floors that travel too quickly. We even built a solar system with gravity, which is no mean feat. Everything, however, has been leading up to one challenge: a final game, which we'll be making in chapters nine and ten. That gives us one last chance to round up all of the loose ends and learn everything we need to make that game in this chapter.

You may wonder what more we could possibly have to learn about? Well, in this piece we're going to cover Python classes. On the surface, this may not sound as exciting as making a solar system, but it is nevertheless an extremely important part of making games, especially in Python. We won't be looking too closely at the code for this particular game; instead, we will concentrate on studying the structures used to make the game work, and use the game code to demonstrate the principle. If you just want to work through the code and get it running for fun, though, you can find all of the code and images you will need on GitHub (**github.com/seanmtracey/Games-with-Pygame**).

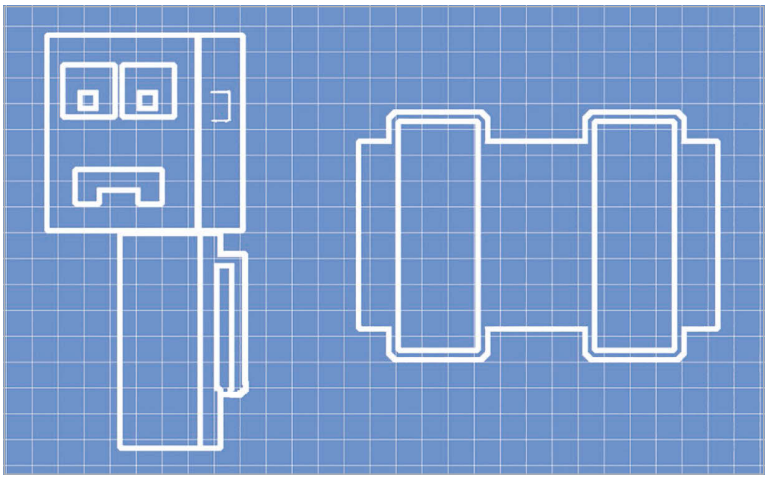
What is a class?

Throughout this series, we've come across many different data types, including variables, lists and dictionaries. Each data type exists to help us organise and store data in a way that makes it easy for us to reuse and reference throughout our games. A class is another such structure that helps us to organise our code and objects. Classes are designed to be a kind of blueprint for bits of code that might need to be reused again and again. Let's compare and contrast classes with dictionaries. To make a dictionary that, let's say, describes a rectangle, we could write something like this:

```
aRectangle = {
                    "x" : 5,
                    "y" : 10,
                    "width" : 20
                    "height" : 30
             }
```

That would certainly do the job; we have everything we need to make a rectangle: width, height, x, and y. It's great, but what if we want to make another rectangle? We could simply write it again:

```
rectangleOne = {
      "x" : 5,
      "y" : 10,
      "width" : 20
      "height" : 30
}

rectangleTwo = {
      "x" : 5,
      "y" : 10,
      "width" : 20
      "height" : 30
}
```



Right Think of classes as a blueprint for things that can be made and used many times

But that's a little messy. Instead, we could create a function to make rectangles for us:

```
def rectangleMaker(width, height, x, y):

    return {
            "x" : x,
            "y" : y,
            "width" : width,
            "height" : height
    }

rectangleOne = rectangleMaker(20, 15, 30, 50)
rectangleOne = rectangleMaker(10, 35, 40, 70)
```

That's better, but in order to make rectangles a more convenient thing to create quickly, we've had to write a function which builds one and passes the new 'rectangle' (it's not really a rectangle, it's a dictionary describing something that *could* be a rectangle) back to whatever bit of code wanted it. This does the job, and that's how we've done things so far, but it's not very semantic. Classes do all

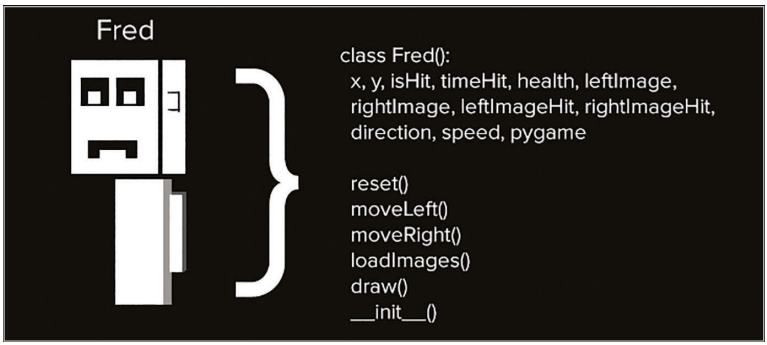
that we just looked at and more: they can keep track of themselves and their properties; they can contain functions that can execute code that affects the properties in a clever way, rather than having to trigger things manually. Classes can also be 'extended'; that is, a class can be made up of several other classes to make something with the properties and abilities of all the other classes before it. If we were to make squares with a class, we would first define what a rectangle is with a class:

```
class Rectangle():

  x = 0
  y = 0
  width = 0
  height = 0

  def __init__(self, x, y, width, height):
    self.x = x
    self.y = y
    self.width = width
    self.height = height
```

Our class **Rectangle** has four properties – x, y, width, and height – just like our dictionaries, but it also has a function, **__init__**. This is a special function; when we want to create a new rectangle, we simply call **Rectangle** and pass it the values we want to use into it:

```
rectangleOne = Rectangle(0, 5, 20, 30)
rectangleTwo = Rectangle(0, 5, 20, 30)
```

When we call **Rectangle** like this, we are triggering something called 'instantiation'. In its simplest terms, instantiation means we're creating something new from a blueprint (our class) that can operate independently from other objects in our code. When we instantiate



Right This is Fred. He's our game avatar and he's made up of a class. There's only one Fred

a new `Rectangle`, that special function `__init__` will be called and will receive the variables we pass through to it. There's something a little different here, though: our `__init__` function is expecting five arguments to be passed to it, but we only passed four, and Python didn't complain. Why is this? When we instantiate a new `Rectangle`, `self` gets passed through to the `__init__` function by the class itself and it refers to its 'self'. With `self`, we can create as many `Rectangles` as we like and have functions inside of the class reference itself, set values, and run code that only affects itself. It's this useful property that makes classes far more useful than standard dictionaries. We can also reference the properties and functions of each instance of our `Rectangles` using '`.`' instead of having to use `[]`.

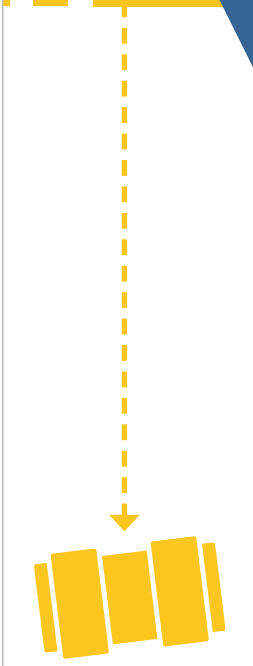
This is Fred

So, now we have a grip on what a class is: it's a blueprint full of code and variables that can be used many times across our Python projects. This makes them great for games, because we can create and control objects dynamically as and when we need them, rather than having to specify everything by hand, like clouds or cars or buildings or trees or any other object which has variety. That said, this doesn't mean we can only use classes when we expect to make more than one of a thing.

Fred is our game avatar. He works from nine to five in a barrel factory that, frankly, flouts health and safety regulations regarding the storing of barrels in overhead containers. Fred is a simple fellow, so much so that we can describe everything about Fred and all he does in a Python class – but there's only one Fred; nobody else would ever agree to the monotonous labour of the barrel factory. Fred is a one-off.

Download
magpi.cc/1jQkU2b

```
class Fred():

  # Fred's preset variables
  x = 0
  y = 625

  isHit = False
  timeHit = 0
  health = 100

  leftImage = None
  rightImage = None
  leftImageHit = None
  rightImageHit = None

  direction = 1
  speed = 8
  pygame = None

  def reset(self, x):
      # Code for getting Fred ready for
      # another day of dodging barrels

  def moveLeft(self, leftBound):
      # Move Fred to the left

  def moveRight(self, rightBound):
      # Move Fred to the right

  def loadImages(self, pygame):
      # Get the image we need to draw Fred

  def draw(self, surface, time):
      # Draw Fred

  def __init__(self, x, pygame, surface):
      # Create Fred and acquaint
      # him with himself
```



'We are all stardust'... except for Fred. He's a class; this is the blueprint for Fred's existence and this is him at his most basic. As we've said, classes are great when you need to control loads of the same but slightly different things, but classes are great for abstracting (or hiding away) bits of code that we use a lot but aren't useful everywhere and aren't used all of the time. So how do we create Fred? Our **Fred** class lives in the **objects.py** file of our projects, so on line 5 of **freds_bad_day.py**, we import objects. This file contains our **Fred** and our **Barrel** class (which we'll look at in a little bit). Once we've imported the **objects** file, we can create Fred with the following code:

```
Fred = objects.Fred(windowWidth / 2)
```

The argument we passed through will set Fred's x coordinate – we always want Fred to start in the middle of the screen, so that's what we've passed through here. Now 'Fred' has been instantiated, using the **Fred** variable, we can access all of Fred's properties and methods. If we want to know where Fred is now, we can simply call:

```
Fred.x
> 500
```

If we want to move Fred to the left, we can just **+=** the **x** property:

```
Fred.x += 5
```

Fred also has a method called **moveLeft()**; instead of manually setting the x coordinate, we can call the **moveLeft** method instead. This looks tidier, makes it obvious what's happening when the code is being read by somebody else, and allows us do more than one thing when we move Fred to the left:

```
Fred.moveLeft()
```

Instead of adding an arbitrary number, `moveLeft()` checks Fred's `speed` property and adds that to Fred's `x` property. It also checks what direction Fred is moving in and sets the `direction` property accordingly. This means when it comes to drawing Fred, we can draw him facing the correct direction without having to manually check which direction he's facing every time.

```
def moveLeft(self, leftBound):

    if self.direction is not 0:
      self.direction = 0

    if((self.x - self.speed) > leftBound):
      self.x -= self.speed
```

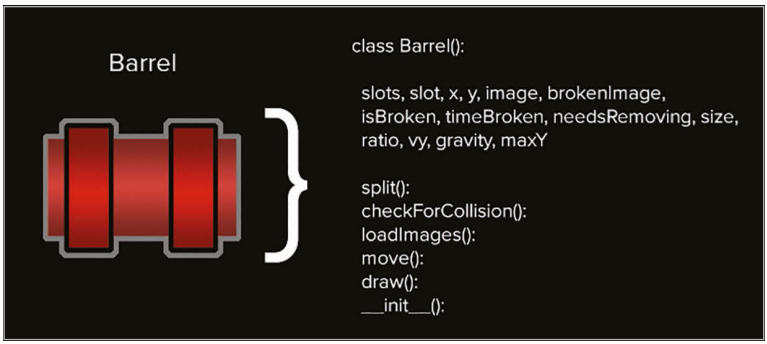
It's simple to use, but it saves us a lot of trouble. In our **freds_bad_day.py** file, the code that actually affects Fred is four lines long:

```
Fred.draw()
Fred.moveLeft()
Fred.moveRight()
Fred.loadImages()
```

That's much simpler than having functions in our code that *could* be accessed by any function but really only need to be used for one object.

This is Fred's nemesis

'Do you know what nemesis means? A righteous infliction of retribution manifested by an appropriate agent.' The common barrel is the blight of Fred's life. He spends his day shift running left to

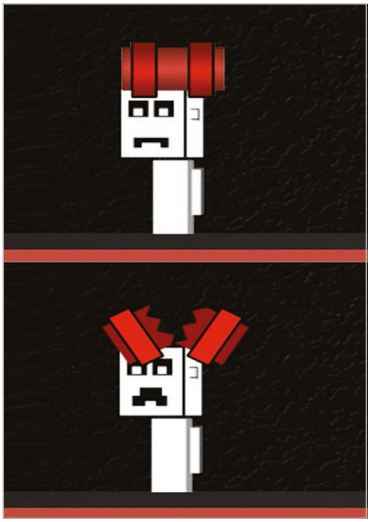


Right This is a barrel. There are many like it, but this one is ours

right and back again, cursing whichever middle-manager it was who thought storing a seemingly unlimited supply of barrels 20 feet above the ground would be a risk-free idea. Unlike Fred, there are many barrels, but at their core, they're both the same. **Fred** and **Barrel** are both classes, but **Fred** is only instantiated once, whereas our **Barrel** is instantiated potentially hundreds of times (depending on how bad Fred's day is).

```
class Barrel():

  slots = [(4, 103), (82, 27), (157, 104), (
234, 27), (310, 104), (388, 27), (463, 104), (
539, 27), (615, 104), (691, 27), (768, 104), (
845, 27), (920, 104)]
  slot = 0
  x = 0
  y = 0

  image = None
  brokenImage = None

  isBroken = False
  timeBroken = 0
  needsRemoving = False

  size = [33,22]
```

```
ratio = 0.66

vy = 1.5
gravity = 1.05
maxY = 20

def split(self, time):
  self.isBroken = True
  self.timeBroken = time
  self.vy = 5
  self.x -= 10

def checkForCollision(self, fred):
    # Check whether barrel is colliding


def loadImages(self, pygame):
    # Load the images for Fred
```



Above A barrel splitting when it hits Fred

BASH! CRASH! THUMP!

Much to Fred's dismay, there's more than one barrel in the world, and we need a place to keep track of them. A new barrel is created after a certain amount of time passes (this amount of time gets smaller as the game progresses), and we append that barrel to the `Barrels` list at the top of our **freds_bad_day.py** file. Every time our 'main' loop works through itself, we iterate through the `Barrels` list, move the barrels, and check if they've hit Fred.We do this with each barrel's `checkForCollision()` method. Our barrel doesn't care about what

it's hitting unless it's Fred: this, however, is very important as the barrel needs to know when it has hit Fred so that it can split in half. One of the clever things about **Fred** (and about classes as a whole) is that once we've instantiated them, we can pass that reference around our code to other classes and functions as we like. When we call **checkForCollisions()**, we pass **Fred** as an argument. This way, the barrel can check it current position and compare it to Fred's. If there's a collision, the barrel will return **True**, then our main loop can take over and split our barrel in half, reduce Fred's health bar, and tell Fred that he should frown – providing Fred is still standing, otherwise the main loop will end the game.

```
hasCollided = barrel.checkForCollision(Fred);

if hasCollided is True:
  barrel.split(timeTick)
  Fred.isHit = True
  Fred.timeHit = timeTick
  if Fred.health >= 10:
    Fred.health -= 10
  else :
      gameOver = True
      gameFinishedTime = timeTick
```

The cleanup

When our barrel has managed to hit poor Fred, it splits in two and continues to fall off the screen. When our barrel goes off screen, we should really delete it, because we don't need it any more and it's eating up our Raspberry Pi's resources. What would be ideal is if our barrel could self-destruct, as it were, and remove itself from our game, but our barrel isn't keeping track of itself inside the game state; the game is keeping track of the barrel in the game and the barrel is looking after itself. This means that once our barrel has split in two, we need a way to reference it so we can delete it. If we simply delete the barrel at the index it's at in the **Barrels** list, all of our remaining barrels will flash while they shift position in the

list. Instead, we add the barrel's index to the `barrelsToRemove` list and once we've finished drawing all of our barrels, we remove all the split barrels before they're drawn again. No mess, just smooth cleanup.

```
    def move(self, windowHeight):
        # Move our barrel

    def draw(self, surface, pygame):
        # Draw our barrel

    def __init__(self, slot):
        # Create a barrel and slot it in the right place
```

The `Barrel` class is simpler than our `Fred` class; this is because it needs to do less. Our barrel only needs to move down and accelerate. When we instantiate `Fred`, we passed through the x coordinate that we wanted Fred to start off from. We don't want our barrels to be able to appear just anywhere along the x/y axis of our game; instead, we want them to appear in one of the 13 slots at the top of our game. Our `Barrel` class has a `slots` list that contains x and y coordinates for each of the barrel holes at the top of our game window. When we want to create a barrel, we don't pass the x coordinate; instead, we pass a random integer between 0 and 12 (remember, computers count from 0, so 0 is the first item in a list) and the coordinates for that slot will become the location for that barrel. All of this happens in that handy `__init__` function.

```
if barrel.needsRemoving is True:
    barrelsToRemove.append(idx)

for index in barrelsToRemove:
  del Barrels[index]
```

Recap

There's been a lot to take in for this chapter, so let's have a recap before we start looking forward building our space shooter game in the final two chapters. We have learned that a class is like a blueprint of code for an object that we want to use. We can use classes once or thousands of times. Using classes helps us keep our code tidy and reusable. When we use a class, we create a new instance of that class which is called 'instantiation'. Each instance of a class can reference itself and so long as it's in the correct scope, any other object can read the properties of our class and can trigger the methods that the class may have. Each method has access to a reference to itself when it's called, even if nothing is passed; this reference will be the first argument passed, but we can pass as many arguments as we want to use, just like any other method.

In chapters nine and ten, we are going to use absolutely everything we've learned so far to make an exciting space shooter game. There will be spaceships, lasers, gravity, sound effects and all sorts of other thrills and spills. You'll love it!

Freds_bad_day.py

Download
magpi.cc/1jQkU2b

```
01. import pygame, sys, random, math
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05. import objects
06.
07. windowWidth = 1000
08. windowHeight = 768
09.
10. pygame.init()
11. pygame.font.init()
12. surface = pygame.display.set_mode((windowWidth,
    windowHeight), pygame.FULLSCREEN)
13.
14. pygame.display.set_caption('Fred\'s Bad Day')
15. textFont = pygame.font.SysFont("monospace", 50)
16.
17. gameStarted = False
18. gameStartedTime = 0
19. gameFinishedTime = 0
20. gameOver = False
21.
```

```
startScreen = pygame.image.load("assets/startgame.png")
endScreen = pygame.image.load("assets/gameover.png")

background = pygame.image.load("assets/background.png")
Fred = objects.Fred(windowWidth / 2)
Barrels = []
lastBarrel = 0
lastBarrelSlot = 0
barrelInterval = 1500

goLeft = False
goRight = False

def quitGame():
  pygame.quit()
  sys.exit()

def newBarrel():
  global Barrels, lastBarrel, lastBarrelSlot

  slot = random.randint(0, 12)

  while slot == lastBarrelSlot:
      slot = random.randint(0, 12)

  theBarrel = objects.Barrel(slot)
  theBarrel.loadImages(pygame)

  Barrels.append(theBarrel)
  lastBarrel = GAME_TIME.get_ticks()
  lastBarrelSlot = slot

Fred.loadImages(pygame)

# 'main' loop
while True:

  timeTick = GAME_TIME.get_ticks()

  if gameStarted is True and gameOver is False:

    surface.blit(background, (0, 0))

    Fred.draw(surface, timeTick)

    barrelsToRemove = []

    for idx, barrel in enumerate(Barrels):
      barrel.move(windowHeight)
```

```
71.         barrel.draw(surface, pygame)
72.
73.         if barrel.isBroken is False:
74.
75.           hasCollided = barrel.checkForCollision(Fred)
76.
77.           if hasCollided is True:
78.             barrel.split(timeTick)
79.             Fred.isHit = True
80.             Fred.timeHit = timeTick
81.             if Fred.health >= 10:
82.               Fred.health -= 10
83.             else :
84.              gameOver = True
85.              gameFinishedTime = timeTick
86.
87.         elif timeTick - barrel.timeBroken > 1000:
88.
89.           barrelsToRemove.append(idx)
90.           continue
91.
92.         if barrel.needsRemoving is True:
93.           barrelsToRemove.append(idx)
94.           continue
95.
96.       pygame.draw.rect(surface, (175,59,59), (
    0, windowHeight - 10, (windowWidth / 100) * Fred.health, 10))
97.
98.       for index in barrelsToRemove:
99.         del Barrels[index]
100.
101.      if goLeft is True:
102.        Fred.moveLeft(0)
103.
104.      if goRight is True:
105.        Fred.moveRight(windowWidth)
106.
107.    elif gameStarted is False and gameOver is False:
108.       surface.blit(startScreen, (0, 0))
109.
110.    elif gameStarted is True and gameOver is True:
111.            surface.blit(endScreen, (0, 0))
112.            timeLasted = (
    gameFinishedTime - gameStartedTime) / 1000
113.
114.            if timeLasted < 10:
115.                 timeLasted = "0" + str(timeLasted)
116.            else :
```

```
                timeLasted = str(timeLasted)

            renderedText = textFont.render(
    timeLasted, 1, (175,59,59))
            surface.blit(renderedText, (495, 430))

    # Handle user and system events
    for event in GAME_EVENTS.get():

      if event.type == pygame.KEYDOWN:

        if event.key == pygame.K_ESCAPE:
          quitGame()
        elif event.key == pygame.K_LEFT:
          goLeft = True
          goRight = False
        elif event.key == pygame.K_RIGHT:
          goLeft = False
          goRight = True
        elif event.key == pygame.K_RETURN:
          if gameStarted is False and gameOver is False:
            gameStarted = True
            gameStartedTime = timeTick
          elif gameStarted is True and gameOver is True:
            Fred.reset(windowWidth / 2)


            Barrels = []
            barrelInterval = 1500

            gameOver = False

    if event.type == pygame.KEYUP:

      if event.key == pygame.K_LEFT:
        goLeft = False
      if event.key == pygame.K_RIGHT:
        goRight = False

      if event.type == GAME_GLOBALS.QUIT:
        quitGame()

    pygame.display.update()

    if GAME_TIME.get_ticks() - lastBarrel > barrelInterval \
and gameStarted is True:
      newBarrel()
      if barrelInterval > 150:
        barrelInterval -= 50
```

Objects.py

Download
magpi.cc/1jQkU2b

```
01. class Fred():
02.
03.   x = 0
04.   y = 625
05.
06.   isHit = False
07.   timeHit = 0
08.   health = 100
09.
10.   leftImage = None
11.   rightImage = None
12.   leftImageHit = None
13.   rightImageHit = None
14.
15.   direction = 1
16.   speed = 8
17.   pygame = None
18.
19.   def reset(self, x):
20.     self.x = x
21.     self.y = 625
22.
23.     self.isHit = False
24.     self.timeHit = 0
25.     self.health = 100
26.
27.     self.direction = 1
28.     self.speed = 8
29.     self.pygame = None
30.
31.   def moveLeft(self, leftBound):
32.
33.     if self.direction is not 0:
34.       self.direction = 0
35.
36.     if((self.x - self.speed) > leftBound):
37.       self.x -= self.speed
38.
39.   def moveRight(self, rightBound):
40.
41.     if self.direction is not 1:
42.       self.direction = 1
43.
44.     if((self.x + self.speed) + 58 < rightBound):
45.       self.x += self.speed
46.
47.   def loadImages(self, pygame):
```

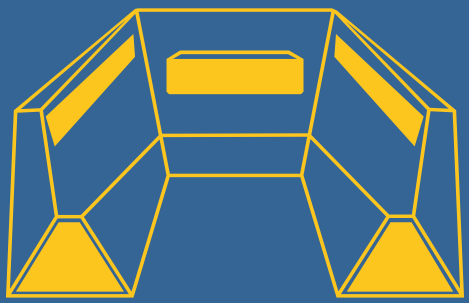
```
48.     self.leftImage = pygame.image.load(
“assets/Fred-Left.png”)
49.     self.rightImage = pygame.image.load(
“assets/Fred-Right.png”)
50.     self.leftImageHit = pygame.image.load(
“assets/Fred-Left-Hit.png”)
51.     self.rightImageHit = pygame.image.load(
“assets/Fred-Right-Hit.png”)
52.
53.   def draw(self, surface, time):
54.
55.     if time - self.timeHit > 800:
56.       self.timeHit = 0
57.       self.isHit = False
58.
59.     if self.direction is 1:
60.       if self.isHit is False:
61.         surface.blit(self.rightImage, (self.x, self.y))
62.       else :
63.         surface.blit(self.rightImageHit, (
self.x, self.y))
64.     else :
65.       if self.isHit is False:
66.         surface.blit(self.leftImage, (self.x, self.y))
67.       else :
68.         surface.blit(self.leftImageHit, (self.x, self.y))
69.
70.   def __init__(self, x):
71.     self.x = x
72.
73. class Barrel():
74.
75.   slots = [(4, 103), (82, 27), (157, 104), (234, 27), (310, 104), (388, 27), (
463, 104), (539, 27), (615, 104), (691, 27), (768, 104), (845, 27), (920, 104)]
76.   slot = 0
77.   x = 0
78.   y = 0
79.
80.   image = None
81.   brokenImage = None
82.
83.   isBroken = False
84.   timeBroken = 0
85.   needsRemoving = False
86.
87.   size = [33,22]
88.   ratio = 0.66
89.
90.   vy = 1.5
91.   gravity = 1.05
```

```
92.    maxY = 20
93.
94.    def split(self, time):
95.      self.isBroken = True
96.      self.timeBroken = time
97.      self.vy = 5
98.      self.x -= 10
99.
100.   def checkForCollision(self, fred):
101.
102.     hitX = False
103.     hitY = False
104.
105.     if fred.x > self.x and fred.x < self.x + 75:
106.       hitX = True
107.     elif fred.x + 57 > self.x and fred.x + 57
    < self.x + 75:
108.       hitX = True
109.     if fred.y + 120 > self.y and fred.y < self.y:
110.       hitY = True
111.     elif fred.y < self.y + 48:
112.       hitY = True
113.     if hitX is True and hitY is True:
114.       return True
115.
116.   def loadImages(self, pygame):
117.     self.image = pygame.image.load("assets/Barrel.png")
118.     self.brokenImage = pygame.image.load(
    "assets/Barrel_break.png")
119.
120.   def move(self, windowHeight):
121.
122.     if self.vy < self.maxY:
123.       self.vy = self.vy * self.gravity
124.     self.y += self.vy
125.
126.     if self.y > windowHeight:
127.       self.needsRemoving = True
128.
129.   def draw(self, surface, pygame):
130.     if self.isBroken is True:
131.       surface.blit(self.brokenImage, (self.x, self.y))
132.     else :
133.       surface.blit(self.image, (self.x, self.y))
134.
135.   def __init__(self, slot):
136.     self.slot = slot
137.     self.x = self.slots[slot][0]
138.     self.y = self.slots[slot][1] + 24
```

The MagPi ESSENTIALS

[CHAPTER NINE]
THE ALIENS ARE TRYING TO KILL ME

In chapter nine, we make the first half of our final game project, putting to use everything we've learned so far.



We have covered a wealth of material on the subject of making games with Pygame, and it is time to put everything we have learned into practice. Over the final two chapters, we are going all of our new knowledge to make a space-shooter game. Using our mouse, we're going to control a small but feisty spaceship which will fend off wave after wave of merciless alien hordes, by blasting them with a lethal green laser-ray. In chapter ten, we will also learn how to make levels with organised structures, instead of the random placement of enemies that we will have in this chapter's version of our game.

We'll also add a game-over screen, some UI elements like health and ammunition counters, and we'll add some shields to our space vessel too, because who doesn't like shields?

Since we're not learning anything new this time, we don't need to explore any abstract game or programming concepts before we can make something; we're just going to walk through the code and figure out what we've done and why we've done it that way. So let's look at the code first, and specifically, at the structure of our code. You may notice that the code for our game is not in one large file as has been the case for most of our previous games. Instead, it has been split across three separate files: one for our main game logic (we've called it **aliens.py**), one that contains the code for our spaceships (**ships.py**), and one file that contains all of the information about our lasers and bullets (**projectiles.py**).

Right The start screen for our final game

Above A screenshot of the game that we'll make in this half of the tutorial

aliens.py is where we will run our game from. It is responsible for handling how we react to user interactions and game events, such as moving and firing the ship, creating new enemies, and triggering sounds. **ships.py** and **projectiles.py** will be imported by **aliens.py**, and will be used to create our own spaceship, the enemy space ships, and the projectiles of both of these types of ship.

Aliens.py

Let's break down the structure of **aliens.py** first. This is where everything in the game will start from, so it makes sense that we should too. As in all of our previous programs, we have our import statements on lines 1–5. Here, we're importing all of the standard Python and Pygame modules that we'll need to make our game do its thing. We also import our own file, **ships.py**, which sits in the same folder as our **aliens.py** folder, with `import ship`.

On lines 7–39, we have all of the global variables that we'll use to keep track of the various objects and events that occur in our game. These are 'global' variables because they don't fall within the scope of any function in our program, which means that that any function in our game can read and change the variables as they like.

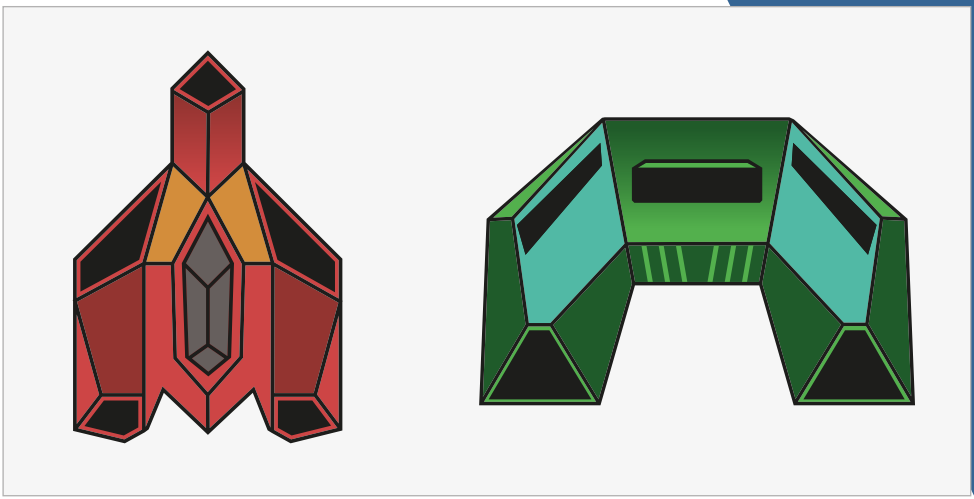
[QUICK TIP]

This is the first of two halves of our final game. Everything works but doesn't look too polished yet, and the game mechanics are simple, but remember: we're setting the stage for chapter 10.

In a lot of circumstances this is frowned upon, but for games it's perfect. Not every variable we need to make this game is declared here; there are quite a few more in our ships and projectile classes which we will get to shortly. Remember, using classes is a great way to keep properties that are relevant to the thing we're using all wrapped up nicely together.

In previous chapters, we've always used a main loop to update our game state and draw the items and objects about the place as we've needed. We're breaking that habit this time. Instead of having the main loop be the sole place where game stuff happens, we're going to create two extra functions and split up the work between them. These two functions are `updateGame()` and `drawGame()`, and their purposes are pretty simple. `updateGame()` (lines 41–75) will always be called before `drawGame()` (lines 77–84); it will update everything that has changed since the last loop through our main function. It will also calculate the new positions and statuses of all of our game objects if they have changed, and will then update those properties. Straight after that, `drawGame()` is called; this is responsible only for drawing the elements of the game that may or may not have been changed. `drawGame()` could update different properties if we wanted it to, but that leads to messy code with multiple places where things can go wrong. This way, if something breaks in our game logic, we'll know that it most probably broke in `updateGame()` and not somewhere else. You'll notice that `drawGame()` is a good deal smaller than `updateGame()`; this is not necessarily because its job is simpler, but because all of the drawing onto our surface has been abstracted away to individual classes. Each object is responsible for drawing itself, but `drawGame()` has to tell them to do it.

Last, but certainly by no means least, we have our 'main loop' on lines 91–135. Now that our main loop is no longer responsible for updating or drawing our game events, it's mainly concerned with detecting the state of our mouse and timing events, like creating a new enemy spaceship after a certain amount of time. In a way, our main loop is still responsible for updating and drawing our game, in that it calls `updateGame()` and `drawGame()` at the correct time based on our game variables; the responsibilities have simply been abstracted so our code is a little more decipherable to people who didn't write the code or read this tutorial.



Above The two ships for our game: our ship (left) and an alien ship (right)

Ships.py

On line 32 of **aliens.py**, we have the variable **`ship`**. This is where we create our player's spaceship. With it, we shall defend the Earth, our solar system and, yes, even the galaxy from the tyranny of kinds of alien evil! This variable instantiates our **`Player`** ship class that we imported from our **ship.py** file. If you take a look at **ship.py**, you'll see it's almost as big as **aliens.py**. That should make sense: after all, spaceships are complicated things. In our **ships.py** file, we have two different classes: our **`Player`** class (remember, class names usually start with a capital letter) and our **`Enemy`** class. The **`Player`** class is where all of the logic for moving, firing, drawing, and damaging our own spaceship happens. We also keep track of the sound effects and images used by our spaceship as we play our game. Almost all of these functions will be called by the code in our **aliens.py** file, but our ship can call functions on itself too. For example, when we initialise our **`Player`** class, we call the **`loadImages()`** function inside of our **`__init__`** function with **`self.loadImages()`** so that we load our ship images straight away, rather than causing a delay when we need to draw our ship for the first time.

Our **`Enemy`** class is much, much smaller than our **`Player`** class. This does not, however, mean it is less complicated. If you look at line 73 of **ships.py**, you'll see that when we call the **`Enemy`** class, we pass through

[QUICK TIP]

The sounds for this tutorial were created using BFXR (**bfxr.net**), a nifty little tool designed for creating sound effects that resemble those from games of times long past. Go and have a play with it!

the `Player` class as an argument. You may be wondering why we do this: although we have not been able to cover it before, it's a really neat thing that classes can do. When we instantiate a class, if we include the name of another class in its declaration when we actually write the code, it will get all of the properties and classes of the class that has been passed through. We do this because, despite being on opposite sides of our epic cosmic war of wits and lasers, at its core, a spaceship is a spaceship like any other but with a few tweaks here and there.

So, even though our `Enemy` class doesn't have `checkForHit`, `registerHit` and `checkForBullets` methods typed out, it still has those methods – it just gets them from `Player`. This enables us to use code in different ways across multiple objects, but we can also overwrite some of those methods and add new ones as they're needed for our `Enemy` class. For example, our `Enemy` class has a `tryToFire()` function. Our `Player` class doesn't have this; only our `Enemy` class does. We can also set different values for the same variables in our `Enemy` class: our `bulletSpeed` value in `Enemy` is 10, whereas it's -10 in our `Player` class. And, of course, the image we're using for each type of ship is different.

Above The two different projectiles for our ship types: our projectile (top) and the alien projectile (bottom)

Projectiles.py

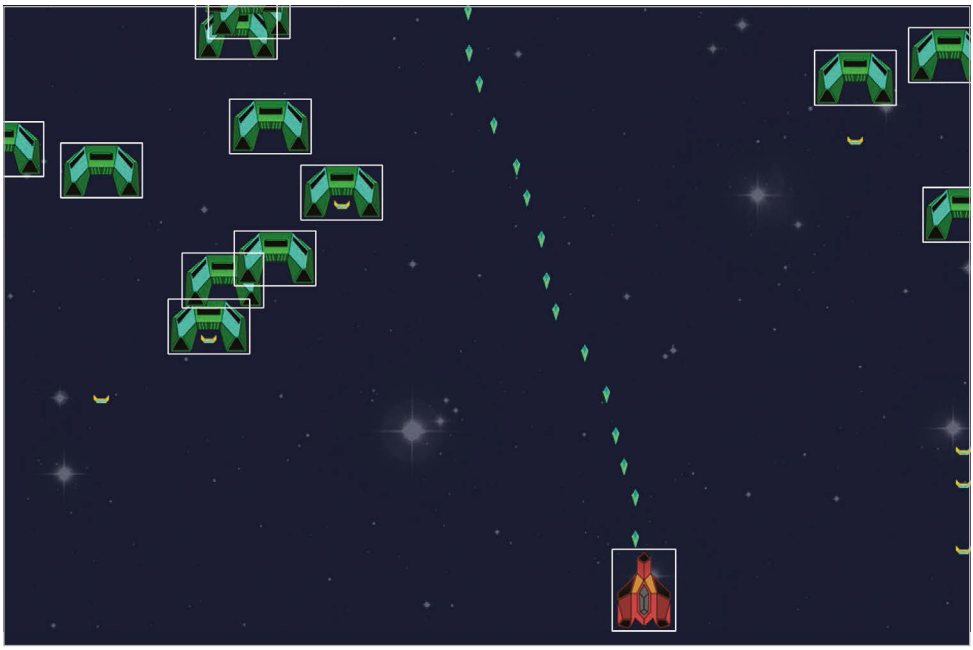
Continuing our use of classes, we have our projectiles `Bullet` class in our **projectiles.py** file. Note that the latter isn't imported into our game in **aliens.py** but in **ships.py**, because our game doesn't need bullets – our ships do. Our `Bullet` class is far simpler than our two ship classes: we have

> The first thing we do on each loop of our game is set three variables...

only three methods and a range of variables to affect and track each bullet: `move`, `draw`, and `loadImages`. How do the bullets know when they are hitting something? Because each bullet created in our game is stored in the `bullets` list in each of our ships, we use the `ships` class method `checkForHit` to see whether or not any of the bullets hit anything. There's no real reason for doing it this way – we could have each bullet be responsible for checking if it hit something – but it does make sense to have each ship keep an eye on whether the bullets it fired hit something.

Game events

Now we know what each file is responsible for in our latest game, we can start walking through how they all come together. As soon as our game is run, the global variables are set and all of the imports between lines 1 and 39 are applied. Let's take a look at our main loop starting on line 90 in `aliens.py`. The first thing we do on each of our game loops is to set three variables: `timeTicks`, `mousePosition`, and `mouseStates`. The `timeTicks` variable keeps track of how many milliseconds have passed since we started the game. We can use this to set variables or create new objects after a set amount of time, as we do on line 128, where we create a new enemy ship after every 1.5 seconds of the game playing. `mousePosition` is where we store the position of the mouse in our game window. Remember, our mouse position is relative to the top-left of our Pygame window. We use the mouse to move our ship around in our game, so we need to keep a constant track of where our mouse is; storing our `mousePosition` at the start of each loop saves us a little time and typing if we need to check the position more than once every loop. `mouseStates` is where we save the 'state' of our mouse buttons, namely which buttons are being pressed down and which ones aren't. We use the left mouse button to fire our weapons and start our game so having that information stored globally means we can check against one variable rather than calling `pygame.mouse.get_pressed()` multiple times.



Above The white boxes around our spaceships are a visual representation of the 'hit test' that we're running to check if a projectile has hit a ship

Starting our game

Right after the first three variables, we come across an if-else statement. This will check whether or not our game has started; after all, we want to have a title and game over screen, so we need to know when to show them. The first check on line 97 will see if we are running a game already. If so, it'll update and then draw the game (we'll go through those shortly); otherwise, we go to the next check in our if-else statement. On line 102, if our game hasn't started yet and it hasn't finished either, then we must have just started the program, so let's draw the start screen. On line 103, we blit our start screen image onto our game surface. This image has a Start button drawn on it, so next, on line 105, we check whether or not the left mouse button has been clicked and if it has, we check to see if the click occurred inside the button on line 107. If both of these conditions are met, our `gameStarted` variable is set to `True` and on the next loop, our game will start: time to kill the alien scourge!

Updating the game state

The game has started, the onslaught begins, but how do we go about saving our kind? Line by line, of course! Lines 41–75 of **aliens.py** contain our `updateGame()` method. Here, we update the position of our ships, the enemy ships, and fire our weapons if we click a mouse. On lines 45–49, we check whether or not our mouse is being clicked. If it is, we fire our weapon, but we don't want our guns to keep firing for as long as we hold down the button; we want to fire on each click, so we set the `mouseDown` variable to `True`. This way, we can be certain that we only fire once per click, not willy-nilly. When we fire our weapon, we play a laser sound effect. It may be true that in space no-one can hear you scream, but in *Star Wars* it's infinitely cooler to have blaster sounds going off all around you. Much like when we add an image to our surface to see it drawn on our screen, we add our sound to our mixer to have it played out through our speakers (on lines 31–33 of **ships.py**).

> Unlike our valiant spaceship there are many, many enemy spaceships

Next, on line 51, we set the position of the ship to match where our mouse is. We subtract half of the width of our ship from our mouse X coordinate, so the ship's centre aligns with our mouse pointer.

That's everything to do with our ship updated, so we can move on to the enemy ships. Unlike our single valiant spaceship there are many, many enemy spaceships. We need to update the position and state of each one of them, so we create a loop on lines 57–72 to handle this. First, we move them. Our enemy spaceships aren't that sophisticated when they move: they're hell-bent on our destruction, so they fly straight at us in order to take a potshot. Next, the enemy spaceships will try to take a shot at us. Why 'try'? Well, our enemies are being controlled by Python, which can fire a great deal quicker than you can. `tryToFire()` is called once per ship in every loop and gives our enemy a 1/100 chance of getting off a shot. That might sound like pretty slim odds for firing at all! But remember, our loop is run 60 times a second, which means that there's a roughly 50–50 chance each enemy will fire a shot every two seconds, so we need to keep our wits about us.

Lines 60–61 are where we check whether any shots our enemies have fired have hit us, and whether any shots we've fired have hit our enemies. The `checkForHit` function call does a couple of things. We pass through the thing we want to check that we hit; if we did, our code in **ships.py** on lines 48–49 will figure that out. If we did hit the enemy, or vice versa, our `checkForHit` function will call the `registerHit()` function on the object that we hit, which decreases the health value of our target. Our enemies have one life, whereas we have five. At the end of our `checkForHit()` function, we return `True` if the health of the ship is 0; that way, if our `shipIsDestroyed == True` or our `enemyIsDestroyed == True`, we can end the game or remove the enemy from our game.

Rather than removing our enemy straight away, we add its index in the `enemyShips` list to the `enemiesToRemove` list. Once we've worked out all of the enemies we need to delete, we iterate through the indexes in the `enemiesToRemove` list and delete them one at a time on lines 74–75.

Drawing our game

As noted previously, the `drawGame()` function is much smaller than the `updateGame()` one. This is because `updateGame()` has done all of the hard work for us. We don't need to worry about moving anything here: everything which needs to be updated, like positions and health, has already been taken care of before we get to `drawGame()`.

The first thing we draw with the latter is the background, because we want everything else to be drawn on top of it. We only have a single player spaceship, so we'll draw that first and then we'll draw all of the bullets that we've fired so far on lines 79–80. Next, we'll draw our enemies. Again, we'll do this in a loop because there's the possibility of there being more than one enemy, and we'll draw the bullets that each ship fires too. It wouldn't be much of an armada if there was only one enemy ship.

What next?

That's the first half of our game. We don't have much in the way of a game-over screen, so we'll cover making one in our final chapter. We'll program a UI for our health, add shields to our ships, and include some explosion effects when either our own ship or an enemy ship is destroyed. We'll also write some code that will create levels (waves) that you can customise to make each game behave any way you like.

Aliens.py

Download
magpi.cc/1jQlptg

```
01. import pygame, sys, random, math
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05. import ships
06.
07. windowWidth = 1024
08. windowHeight = 614
09.
10. pygame.init()
11. pygame.font.init()
12. surface = pygame.display.set_mode((windowWidth, windowHeight))
13.
14. pygame.display.set_caption('Alien\'s Are Gonna Kill Me!')
15. textFont = pygame.font.SysFont("monospace", 50)
16.
17. gameStarted = False
18. gameStartedTime = 0
19. gameFinishedTime = 0
20. gameOver = False
21.
22. # Mouse variables
23. mousePosition = (0,0)
24. mouseStates = None
25. mouseDown = False
26.
27. # Image variables
28. startScreen = pygame.image.load("assets/start_screen.png")
29. background = pygame.image.load("assets/background.png")
30.
31. # Ships
32. ship = ships.Player(windowWidth / 2, windowHeight, pygame, surface)
33. enemyShips = []
34.
35. lastEnemyCreated = 0
36. enemyInterval = random.randint(1000, 2500)
37.
38. # Sound setup
39. pygame.mixer.init()
40.
41. def updateGame():
42.
43.   global mouseDown, gameOver
44.
```

```
45.   if mouseStates[0] is 1 and mouseDown is False:
46.     ship.fire()
47.     mouseDown = True
48.   elif mouseStates[0] is 0 and mouseDown is True:
49.     mouseDown = False
50.
51.   ship.setPosition(mousePosition)
52.
53.   enemiesToRemove = []
54.
55.   for idx, enemy in enumerate(enemyShips):
56.
57.     if enemy.y < windowHeight:
58.       enemy.move()
59.       enemy.tryToFire()
60.       shipIsDestroyed = enemy.checkForHit(ship)
61.       enemyIsDestroyed = ship.checkForHit(enemy)
62.
63.       if enemyIsDestroyed is True:
64.         enemiesToRemove.append(idx)
65.
66.       if shipIsDestroyed is True:
67.         gameOver = True
68.         print "\n\n\nYou Died\n\n\n"
69.         quitGame()
70.
71.     else:
72.       enemiesToRemove.append(idx)
73.
74.   for idx in enemiesToRemove:
75.     del enemyShips[idx]
76.
77. def drawGame():
78.     surface.blit(background, (0, 0))
79.     ship.draw()
80.     ship.drawBullets()
81.
82.     for enemy in enemyShips:
83.       enemy.draw()
84.       enemy.drawBullets()
85.
86. def quitGame():
87.   pygame.quit()
88.   sys.exit()
89.
90. # 'main' loop
91. while True:
```

```
92.
93.     timeTick = GAME_TIME.get_ticks()
94.     mousePosition = pygame.mouse.get_pos()
95.     mouseStates = pygame.mouse.get_pressed()
96.
97.     if gameStarted is True and gameOver is False:
98.
99.       updateGame()
100.      drawGame()
101.
102.    elif gameStarted is False and gameOver is False:
103.      surface.blit(startScreen, (0, 0))
104.
105.      if mouseStates[0] is 1:
106.
107.        if mousePosition[0] > 445 and mousePosition[0] < 580 and
      mousePosition[1] > 450 and mousePosition[1] < 510:
108.
109.          gameStarted = True
110.
111.      elif mouseStates[0] is 0 and mouseDown is True:
112.        mouseDown = False
113.
114.    elif gameStarted is True and gameOver is True:
115.            surface.blit(startScreen, (0, 0))
116.            timeLasted = (
      gameFinishedTime - gameStartedTime) / 1000
117.
118.    # Handle user and system events
119.    for event in GAME_EVENTS.get():
120.
121.      if event.type == pygame.KEYDOWN:
122.
123.        if event.key == pygame.K_ESCAPE:
124.          quitGame()
125.
126.    if GAME_TIME.get_ticks()-lastEnemyCreated >enemyInterval and gameStarted is True:

127.      enemyShips.append(ships.Enemy(
      random.randint(0, windowWidth), -60, pygame, surface, 1))
128.      lastEnemyCreated = GAME_TIME.get_ticks()
129.
130.    if event.type == GAME_GLOBALS.QUIT:
131.      quitGame()
132.
133.    pygame.display.update()
134.
```

Ships.py

Download
magpi.cc/1jQlptg

```
01. import projectiles, random
02.
03. class Player():
04.
05.   x = 0
06.   y = 0
07.   firing = False
08.   image = None
09.   soundEffect = 'sounds/player_laser.wav'
10.   pygame = None
11.   surface = None
12.   width = 0
13.   height = 0
14.   bullets = []
15.   bulletImage = "assets/you_pellet.png"
16.   bulletSpeed = -10
17.   health = 5
18.
19.   def loadImages(self):
20.     self.image = self.pygame.image.load("assets/you_ship.png")
21.
22.   def draw(self):
23.     self.surface.blit(self.image, (self.x, self.y))
24.
25.   def setPosition(self, pos):
26.     self.x = pos[0] - self.width / 2
27.     # self.y = pos[1]
28.
29.   def fire(self):
30.     self.bullets.append(projectiles.Bullet(
    self.x + self.width / 2, self.y, self.pygame, self.surface,
    self.bulletSpeed, self.bulletImage))
31.     a = self.pygame.mixer.Sound(self.soundEffect)
32.     a.set_volume(0.2)
33.     a.play()
34.
35.
36.   def drawBullets(self):
37.     for b in self.bullets:
38.       b.move()
39.       b.draw()
40.
41.   def registerHit(self):
42.     self.health -= 1
43.
44.   def checkForHit(self, thingToCheckAgainst):
45.     bulletsToRemove = []
46.
```

```
    for idx, b in enumerate(self.bullets):
      if b.x > thingToCheckAgainst.x and b.x < thingToCheckAgainst.x + thingToCheckAgainst.width:
        if b.y > thingToCheckAgainst.y and b.y < thingToCheckAgainst.y + thingToCheckAgainst.height:
          thingToCheckAgainst.registerHit()
          bulletsToRemove.append(idx)

    for usedBullet in bulletsToRemove:
      del self.bullets[usedBullet]

    if thingToCheckAgainst.health <= 0:
      return True

  def __init__(self, x, y, pygame, surface):
    self.x = x
    self.y = y
    self.pygame = pygame
    self.surface = surface
    self.loadImages()

    dimensions = self.image.get_rect().size
    self.width = dimensions[0]
    self.height = dimensions[1]

    self.x -= self.width / 2
    self.y -= self.height + 10

class Enemy(Player):

  x = 0
  y = 0
  firing = False
  image = None
  soundEffect = 'sounds/enemy_laser.wav'
  bulletImage = "assets/them_pellet.png"
  bulletSpeed = 10
  speed = 2

  def move(self):
    self.y += self.speed

  def tryToFire(self):
    shouldFire = random.random()

    if shouldFire <= 0.01:
      self.fire()

  def loadImages(self):
    self.image = self.pygame.image.load("assets/them_ship.png")

```

```
    def __init__(self, x, y, pygame, surface, health):
      self.x = x
      self.y = y
      self.pygame = pygame
      self.surface = surface
      self.loadImages()
      self.bullets = []
      self.health = health

      dimensions = self.image.get_rect().size
      self.width = dimensions[0]
      self.height = dimensions[1]

      self.x -= self.width / 2
```

Projectiles.py

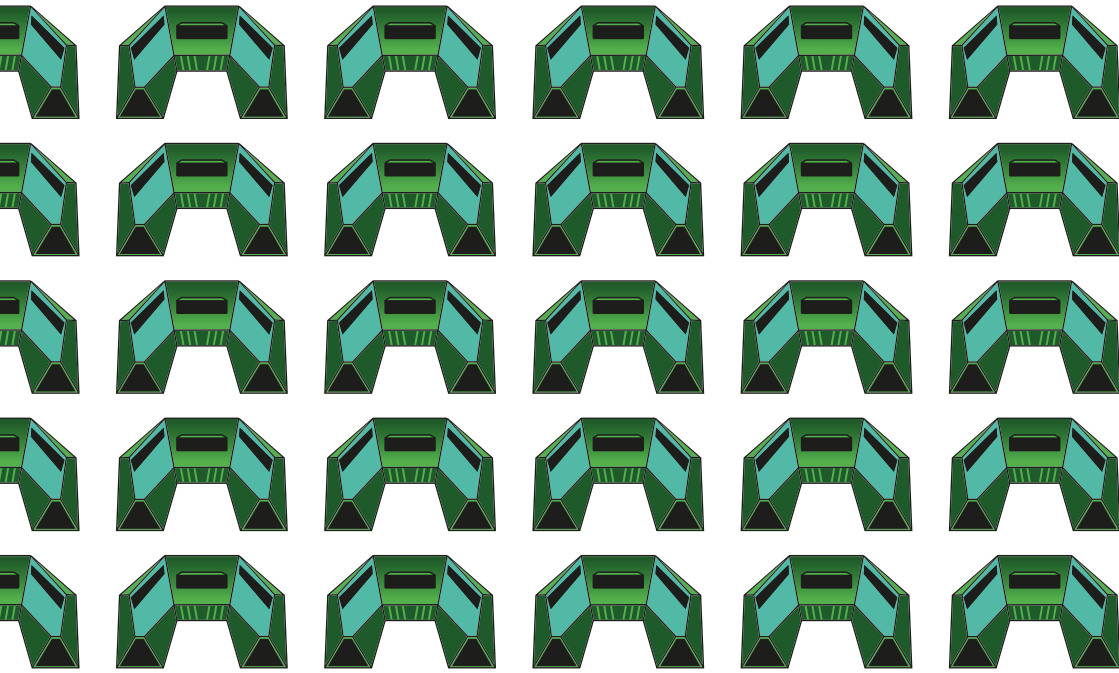
Download
magpi.cc/1jQlptg

```
class Bullet():

    x = 0
    y = 0
    image = None
    pygame = None
    surface = None
    width = 0
    height = 0
    speed = 0.0

    def loadImages(self):
      self.image = self.pygame.image.load(self.image)

    def draw(self):
      self.surface.blit(self.image, (self.x, self.y))

    def move(self):
        self.y += self.speed

    def __init__(self, x, y, pygame, surface, speed, image):
      self.x = x
      self.y = y
      self.pygame = pygame
      self.surface = surface
      self.image = image
      self.loadImages()
      self.speed = speed

      dimensions = self.image.get_rect().size
      self.width = dimensions[0]
      self.height = dimensions[1]

      self.x -= self.width / 2
```

The MagPi ESSENTIALS

[CHAPTER TEN]
THE ALIENS ARE HERE & THEY'RE COMING IN WAVES!

In the final chapter, we're going to give the space shooter game we started in the last chapter some extra polish.



Welcome to the final chapter! If you have worked this far through the book, you can consider yourself to be quite an expert in building games with Pygame. We are going to round things off by adding a final polish to the space shooter game we began in chapter nine.

If you look over the code from the previous chapter and compare it to the code for this one, you'll see that, despite having the same foundations, there's quite a bit more going on in the code this time around. Previously, we dealt with creating a start screen, moving our ship, firing our weapons, creating enemies, having them fire their weapons, and then removing them from time and space whenever we hit one another. Now, we are going to enrich our game by adding shields to our spaceship and create a simple health/shield bar to show their status. We're also going to write some code that lets us create levels and waves for our enemy spaceships to fall into, as well as writing some logic for announcing that the next level of bad guys are on their way. Finally, we'll create two end screens: one for if the aliens kill us, another for if we survive all of the levels of the onslaught.

A tour at warp speed

We've seen most of this code before, obviously, but a good deal has changed, so we're going to zip through where those changes are before we look at them in more detail.

Let's begin by looking at the key differences in **aliens.py**. On line 7, we now import another file, **gameLevels.py**. This file contains a list with a number of dictionaries which we'll use to place our enemies in the different levels of our game. It's a big file, but it's not a complicated one, and we'll take a look at it shortly.

On lines 24–29, we have some new variables. We will use these to keep track of our game's progress and state, as well as changing levels.

On lines 39–42, we load a couple of extra images to use in our game; these will be our game over and wave announcement graphics.

We've done away with the `lastEnemyCreated` variable we used last time to generate enemies after a certain time interval. Instead, we now have a `launchWave` method that will unleash a group of enemy spaceships, based on the pattern we pass it from our **gameLevels.py** file. `updateGame` has changed quite a bit from last time, but that's because we're just making it a little better at what it does already.

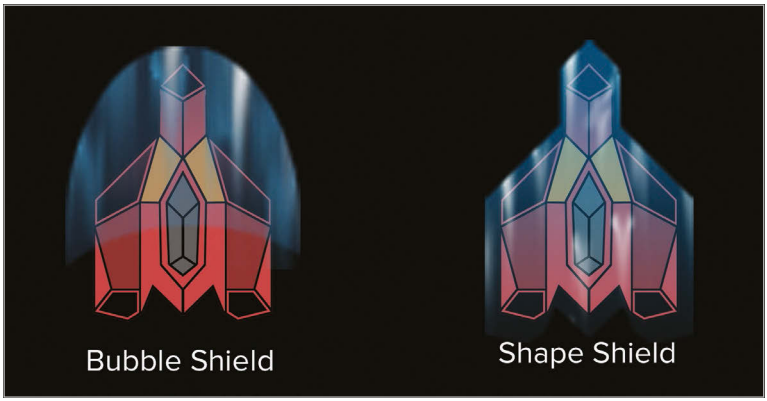
On lines 151 and 152 we now draw a bar across the bottom of our game screen, showing our shields and health status. As either our shields or health decrease, these bars will shrink.

In **ships.py**, we have not changed very much at all; we've just added a tiny bit of logic that creates and draws some basic shields for us. This will be the first thing we take a look at in detail, so we won't examine it here.

Finally, in **projectiles.py**, we've added a `checkForHit` function. You may be wondering why we need this function in the code for our projectiles, when we already have it in the code for our spaceship. Well, you might have noticed in the previous game that when you destroyed an enemy spaceship, the lasers it had fired would disappear along with the ship. This was because each ship was responsible for looking after and updating the bullets it had fired: if there was no ship, there were no bullets. With this function, and a little bit of code in **aliens.py**, we can give the projectiles a place to live after their ship has been destroyed, so that they may strike our spaceship.

[QUICK TIP]

If you know how to use a diff tool, comparing the aliens.py code from chapter ten with chapter nine will help you get a complete oversight of what we've changed and why.



Right There are two different types of shield for this game: a bubble shield, like the USS Enterprise has, or a shape shield (which we think looks cooler), like those found in Stargate. You can change them by loading the image you prefer in the ships class

Full power to the forward deflector shields!

Let's start with the simple things first: what does every spaceship need? Obviously, it has to have energy shields to keep it safe from cosmic dust and enemy fire alike.

Implementing shields for our ship is not partiularly difficult. We already did most of the work already when we created health for our ships. If you look at lines 9, 10, 21 and 22 of **ships.py**, you'll see we've created four new variables. **shieldImage** is where we'll load the image that we'll use to draw our shield. This is simply a transparent PNG which we draw over our ship when it's been hit, to give a cool bit of feedback to our players. **drawShield** is set to True whenever our ship is hit, so we can draw the shield only when we need to instead of the whole time. **shields** and **maxShields** are next. **shields** is the current amount of shield strength we have, and **maxShields** is the maximum amount of shield energy we're allowed to have. Every time our shields are hit, we decrease the shield energy by 1, so we can sustain three hits to our shields before our ship starts to take damage. To let our shields take the brunt of enemy fire before they start to damage our ship, we've tweaked our register hit function inside of our **ship** class. Previously, our **registerHit** method, when called, would decrement (decrease by one) our health value until it was 0. Now, it will check if we have any shield energy left; if our shield levels are greater than 0, we'll decrement the shield level instead of the health level, and we'll set our **drawShield** variable to True so we can draw the shields. If our shields are at 0, then we decrease the health value

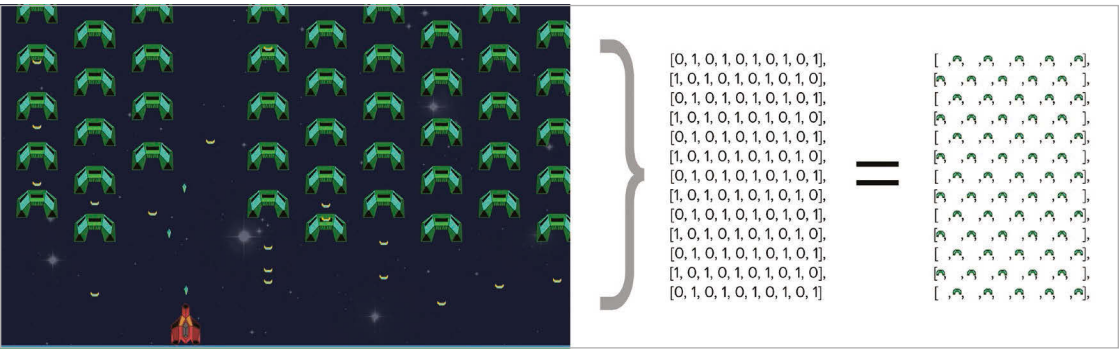
just like we did before. When our `ships draw()` method is called, it will check whether or not our `drawShield` function is True. If it is, we must have taken a hit, so we'll draw the shield and then set `drawShield` to False so it will disappear until we're hit again.

While we're on the subject of shields and health, let's look at where we create health and shield bars. Back in **aliens.py** on lines 151 and 152, we draw a rectangle for our health and another for our shields. It's quite a long line of code, but it's quite simple really. For shields, we take the maximum amount possible that a shield can be (3) and divide the width of the game window (`gameWindow`) by it; we then multiply that value by the current shield level. This gives us the width that our shield bar needs to be to show the amount of energy we have left. If we can sustain three more hits, the bar will be full across the width of our game screen; if it can take two hits, it will fill two-thirds of the screen, and so on until it is empty. We do the exact same thing for our health bar; we just don't affect the values until our shields are depleted.

Let's take a look at a matrix

The biggest change in this version of our game is that we can now define levels and formations for our enemy ships to attack us with. If you take a look at **gameLevels.py**, you'll see there is only one variable, `level`. This is a dictionary that contains objects which describe our levels. Our first level or 'wave' is the first dictionary, the second level is the second dictionary, and so on. Each `level` dictionary has two properties: `interval` and `structure`. Let's take a look at the `structure` property first: this is a list of lists, and in each list is a series of 1s and 0s. Each list is a wave. Think of `structure` as a map of our game window. The width of our game window is represented by one list inside of `structure`. For each 1 in our list, we want to create an enemy spaceship in corresponding space in our game window, and for every 0 we don't. Using this approach, we can define levels

Below By adjusting patterns and intervals, you can make levels completely different from one another with very little effort



Above A visual representation of how we can determine enemy spaceship positions using a matrix

of different difficulty and appearance, just by changing the values of `structure`. For example, if we wanted to create ten ships that spanned the width of the screen at equal intervals, we'd add a list like this to our `structure` list:

```
[1,1,1,1,1,1,1,1,1,1]
```

If we wanted that row of ships to be followed by a row of six ships with a gap in the middle, we'd add two lists to `structure`, one for the first row of spaceships, and another for the second:

```
[1,1,1,1,1,1,1,1,1,1],
[1,1,1,0,0,0,0,1,1,1]
```

This structure is known as a 'matrix', which you can think of as a list with an X and Y axis. If you wanted to know whether or not we were going to create a spaceship in the second grid down from the top of our screen and three across, you could check with `levels[0].structure[2][3]`, but that's not quite how we're using this in our game.

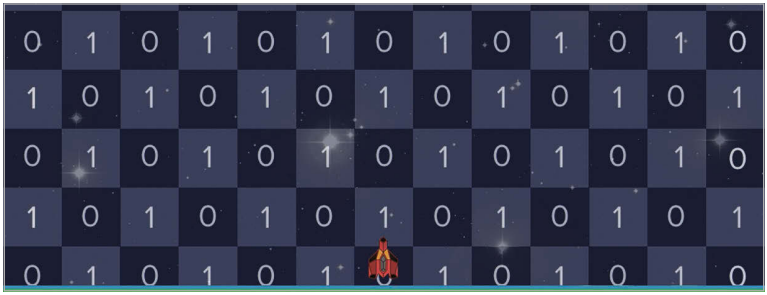
The other property of our level objects is `interval`. This value sets how many seconds should pass before we move on from one wave and create the next. Tweaking this value can greatly change the difficulty of

each level. For example, if you have five waves of enemies with a five second interval between each row, you'll have ten spaceships being generated every five seconds that you need to destroy. That's quite a lot, but 50 enemies over 25 seconds is pretty easy to deal with. However, if you create ten rows of ships and set the interval between waves to two seconds, you're going to be dealing with 100 ships in 20 seconds. That really is an onslaught! To illustrate this, the final level included in **gameLevels.py** contains five times the number of waves than any other level, but there is only one ship in each wave and the interval is 0.5 seconds. This creates a zigzag pattern of ships, which makes for interesting gameplay when you're being fired at by ships across both the X and Y axes. With this knowledge, you can create a limitless variety of levels for our game; all you have to do is copy one of the level objects, and edit the structure and interval as you see fit. That's much more fun than spawning enemies at random X/Y coordinates.

Launch wave!

Now that we know how to structure our levels, how do we put them together in our game? Back in **aliens.py** we have the `launchWave()` function. This takes the current level and wave, and generates enemies at the correct positions and correct time for our game. At the end of our 'main' loop, we have a tiny bit of code on lines 222 and 223 that checks how long it has been since a wave of enemies has been spawned. If more time has passed than is allowed by our interval value for the current level we're playing, `launchWave` is called.

The first thing `launchWave` does is create the variable `thisLevel`. We don't absolutely have to do this, but it makes what we're trying



Left A visual representation of how we map the matrix to the dimensions of our game window

[QUICK TIP]

Just because you're following a tutorial, it doesn't mean you have to use all of the resources we provide. Why not tweak some of the images to create your own unique spaceship? Or mess around with the level structures and ships classes to create more than one enemy ship? Learning comes from trying these things out and seeing how far you get!

to do a little more obvious when we access our level structure, rather than using `gameLevels.level[currentLevel]["structure"]` every time. Next, we check that the wave we're about to create actually exists in our level. If our level has four waves and we try to access a fifth one, our game will crash. If the wave we want to access does exist, we take that wave and assign it to the `thisWave` value. Again, this just makes our code a little nicer to read. We then work through the values of that wave.

First, on line 64 of **aliens.py**, we check whether or not we want to place an alien spaceship here (1) or not (0). If we do, we then do a little bit of maths to work out where to place it. First, we divide the `windowWidth` by the length of the list that makes up the wave; we then place the ship at the X coordinates that are `windowWidth /` the number of slots in the wave * the index of this ship. So, if we have ten

> The first thing launchWave does is create the variable thisLevel....

slots in this wave, and each one wants to have a ship created in each slot, `launchWave` will create ten enemies at equal distances across the width of the game screen. If every other slot in that ten was a 0 and didn't want to have a ship drawn, then five ships would be drawn at equal intervals across the game screen.

Once `launchWave` has created the enemies it needs to, it lets the game continue on its way until it is called by our main loop again. If the next time `launchWave` is run, it finds that there are no more waves in this level, it will check to see if there's another level it can move on to. If so, it will recharge our ship's shields, increase the level number, and reset the wave number to 0. New level! If `launchWave` finds that it's run out of waves to create and that there aren't any more levels to play, it assigns the `gameWon` variable to True. This is a preliminary value, as nothing will happen until all of the enemies have been destroyed, either by our bodacious laser blasts or by them simply flying off the screen to their oblivion. If we survive all of the levels and aren't destroyed by a lucky potshot from one of our alien foes, then we've won the game! Hurrah!

Other tweaks and tidbits

We mentioned earlier that we'd added a `checkForHit` function to our **projectiles.py** `Bullet` class. Let's talk about why we need this for a moment. In a perfect world, where everyone owns a Raspberry Pi and wants to use it exclusively for making Pygame games, every object in a game is responsible for handling itself inside of the larger context of the game. When a projectile hits a target, it runs all of the code it needs to run and then removes itself from the game. Unfortunately, the world is not an ideal place; each object in our Pygame shooter is aware of themself and can interact with the things we tell them to, but they can't be responsible for removing themselves from the game when they're no longer useful, so we need a little bit of code that decides for us. In chapter nine, we made each ship responsible for each bullet that it fired; as such, each ship kept a reference to all the projectiles it had fired in a list accessed with `self.bullets`. Each ship was responsible for cleaning up its own mess, which seems like a reasonable solution. However, this only works as long as there is a ship to clean up after itself. But we destroy spaceships – Earth needs defending, and if nobody else is going to do it, we've got to step up – and when we destroy the enemy spaceships, what happens to bullets they're responsible for? As it stands, they get destroyed too, which doesn't make sense because the bullet left the ship a long time ago. It's a bit like a car catching fire in a car park because there's a blaze in the kitchen back home: the two are related, but not inextricably connected.

YOU SURVIVED THE ALIEN HOARDES
SEE THIS?
THIS IS THE SHIP THAT HEROES FLY
WE SALUTE YOU, SIR!
FANCY ANOTHER GO?
(PRESS SPACE)
THE ALIENS HAVE KILLED YOU
THE CHARRED DEBRIS OF YOUR SHIP ARE SCATTERED
TO FLOAT AMONG THE COSMIC WINDS FOR ETERNITY
WHILE THE MERICILESS ALIEN FORMS BEGIN THEIR
RUTHLESS ANNIHILATION OF HUMANITY
TRY AGAIN?
(PRESS SPACE)

Below The two different game over screens: one for victory and one we hope we never see

So how do we go about solving this? By taking over responsibility for orphaned projectiles, of course! On line 48 of **aliens.py**, we have `leftOverBullets`. This is an empty list. When an enemy is removed from our game (because we've destroyed it or for some

other reason), just before we delete the reference, and any reference to the enemy ship's projectiles entirely, on lines 114–117 of **aliens.py** we go through the bullets array of each enemy we're about to delete and append a reference to each bullet to our `leftOverBullets` list.

> Now, when we delete our enemy ship, we can still animate, move and update the bullets each ship has fired

Now, when we delete our enemy ship, we can still animate, move, and update the bullets each ship has fired, by iterating through the `leftOverBullets` list and checking whether or not our stray projectiles have hit anything. This leads us back to `checkForIt` in the `Bullet` class. With the enemy ship removed from the game, so too is the method we were using to detect the hits. By including a simple version in `Bullet` that returns True or False when a collision is or isn't detected, we can continue to use our game without having to change a great deal of the logic.

As mentioned previously, we have more than one game over screen: one for if we're victorious against the alien scourge, and one for if we're not so fortunate. In order to know which one to show at the completion of the game/demise of our ship, we have a new variable: `gameWon`. This is set to True when our `launchWave` function works out that there are no more waves/levels for it to create. Setting `gameWon` to True is not cause to consider the game 'won' – even though there are no more new enemies to create, there may still be some left over that could destroy our player, so it's not until both `gameWon` is True and the number of `enemyShips` is 0 that we show the congratulations screen. If we're destroyed before the number of enemy ships is 0, `gameWon` is set to False and `gameOver` is set to True, meaning we show the loser screen.

One final note: on line 15, we have our familiar old surface statement – except, if you compare it to previous versions, it's got an extra argument, `pygame.FULLSCREEN`. You can no doubt guess what that does!

That's all, folks!

And that's it: we're finished! You should now be able to go out into the world and make simple video games using Python, a Raspberry Pi, and Pygame. Let's quickly go through all of the things we've learned over the course of this volume. We've learned how to draw basic shapes; how to use a keyboard and mouse to move, create, and delete things; we've learned all about gravity (or at least, a super-simple version of it); we've learned how to bounce things off of other things and how to register things hitting one another; we've learned all about playing sounds and blitting images; and tons and tons of stuff about Pygame and system events. We've also learned that Python is straightforward, and ideal for getting up and going from scratch, for beginners and experts alike. We hope you have enjoyed learning all these new skills, and are looking forward to putting them into practice. Have fun!

levels.py

Download
magpi.cc/1jQlwov

```
level = [
    {
        "structure" :[  [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                    ],
        "interval" : 4
    },
    {
        "structure" :[  [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
```

```
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
                        [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                ],
        "interval" : 5
    },
    {
        "structure" :[  [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
                        [0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
                ],
        "interval" : 4
    },
    {
        "structure" :[  [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
                        [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
```

```
73.                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
74.                 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
75.                 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
76.                 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
77.                 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
78.                 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
79.                 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
80.                 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
81.                 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
82.                 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
83.                 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
84.                 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
85.                 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
86.                 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
87.                 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
88.                 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
89.                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
90.                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
91.                 [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
92.                 [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
93.                 [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
94.                 [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
95.                 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
96.                 [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
97.                 [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
98.                 [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
99.                 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
100.                [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
101.                [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
102.                [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
103.                [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
104.                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
105.                [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
106.                [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
107.                [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
108.                [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
109.                [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
110.                [0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
111.                [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
112.                [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
113.                [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
114.                [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
115.                [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
116.                [0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
117.                [1, 0, 0, 0, 0, 0, 0, 0, 0, 0],
118.                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
119.                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
120.                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
121.                [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
                    ],
            "interval" : 4
        },
        {
            "structure" :[  [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
                            [0, 1, 0, 0, 0, 0, 0, 0, 1, 0],
                            [0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
                            [0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
                            [0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
                    ],
            "interval" : 3
        }
    ]
```

ships.py

Download
magpi.cc/1jQlwov

```
01. import projectiles, random
02.
03. class Player():
04.
05.   x = 0
06.   y = 0
07.   firing = False
08.   image = None
09.   shieldImage = None
10.   drawShield = False
11.   soundEffect = 'sounds/player_laser.wav'
12.   pygame = None
13.   surface = None
14.   width = 0
15.   height = 0
16.   bullets = []
17.   bulletImage = "assets/you_pellet.png"
18.   bulletSpeed = -10
19.   health = 5
20.   maxHealth = health
21.   shields = 3
22.   maxShields = shields
23.
24.   def loadImages(self):
25.     self.image = self.pygame.image.load("assets/you_ship.png")
        self.shieldImage = self.pygame.image.load("assets/shield2.png")

27.
28.   def draw(self):
29.     self.surface.blit(self.image, (self.x, self.y))
30.     if self.drawShield == True:
          self.surface.blit(self.shieldImage, (self.x - 3,
31. self.y - 2))
32.       self.drawShield = False
33.
34.   def setPosition(self, pos):
35.     self.x = pos[0] - self.width / 2
36.
37.   def fire(self):
        self.bullets.append(projectiles.Bullet(self.x + self.width / 2, self.y,
    self.pygame, self.surface, self.bulletSpeed, self.bulletImage))
```

```
38.     a = self.pygame.mixer.Sound(self.soundEffect)
39.     a.set_volume(0.2)
41.     a.play()
42.
43.   def drawBullets(self):
44.     for b in self.bullets:
45.       b.move()
46.       b.draw()
47.
48.   def registerHit(self):
49.     if self.shields == 0:
50.       self.health -= 1
51.     else :
52.       self.shields -= 1
53.       self.drawShield = True
54.
55.   def checkForHit(self, thingToCheckAgainst):
56.     bulletsToRemove = []
57.
58.     for idx, b in enumerate(self.bullets):
59.       if b.x > thingToCheckAgainst.x and b.x <
    thingToCheckAgainst.x + thingToCheckAgainst.width:
60.         if b.y > thingToCheckAgainst.y and b.y <
    thingToCheckAgainst.y + thingToCheckAgainst.height:
61.           thingToCheckAgainst.registerHit()
62.           bulletsToRemove.append(idx)
63.     bC = 0
64.     for usedBullet in bulletsToRemove:
65.       del self.bullets[usedBullet - bC]
66.       bC += 1
67.
68.     if thingToCheckAgainst.health <= 0:
69.       return True
70.
71.   def __init__(self, x, y, pygame, surface):
72.     self.x = x
73.     self.y = y
74.     self.pygame = pygame
75.     self.surface = surface
76.     self.loadImages()
77.
78.     dimensions = self.image.get_rect().size
79.     self.width = dimensions[0]
80.     self.height = dimensions[1]
```

```
81.
82.       self.x -= self.width / 2
83.       self.y -= self.height + 10
84.
85.   class Enemy(Player):
86.
87.     x = 0
88.     y = 0
89.     firing = False
90.     image = None
91.     soundEffect = 'sounds/enemy_laser.wav'
92.     bulletImage = "assets/them_pellet.png"
93.     bulletSpeed = 10
94.     speed = 4
95.     shields = 0
96.
97.     def move(self):
98.       self.y += self.speed
99.
100.    def tryToFire(self):
101.      shouldFire = random.random()
102.
103.      if shouldFire <= 0.01:
104.        self.fire()
105.
106.    def loadImages(self):
107.      self.image = self.pygame.image.load("assets/them_ship.png")
108.
109.    def __init__(self, x, y, pygame, surface, health):
110.      self.x = x
111.      self.y = y
112.      self.pygame = pygame
113.      self.surface = surface
114.      self.loadImages()
115.      self.bullets = []
116.      self.health = health
117.
118.      dimensions = self.image.get_rect().size
119.      self.width = dimensions[0]
120.      self.height = dimensions[1]
121.
122.      self.x += self.width / 2
```

aliens.py

Download
magpi.cc/1jQlwov

```
01. import pygame, sys, random, math
02. import pygame.locals as GAME_GLOBALS
03. import pygame.event as GAME_EVENTS
04. import pygame.time as GAME_TIME
05. import ships
06.
07. import gameLevels
08.
09. windowWidth = 1024
10. windowHeight = 614
11. timeTick = 0
12.
13. pygame.init()
14. pygame.font.init()
15. surface = pygame.display.set_mode((windowWidth, windowHeight),
    pygame.FULLSCREEN|pygame.HWSURFACE|pygame.DOUBLEBUF)
16.
17. pygame.display.set_caption('Alien\'s Are Gonna Kill Me!')
18. textFont = pygame.font.SysFont("monospace", 50)
19.
20. gameStarted = False
21. gameStartedTime = 0
22. gameFinishedTime = 0
23. gameOver = False
24. gameWon = False
25.
26. currentLevel = 0
27. currentWave = 0
28. lastSpawn = 0
29. nextLevelTS = 0
30.
31. # Mouse variables
32. mousePosition = (0,0)
33. mouseStates = None
34. mouseDown = False
35.
36. # Image variables
37. startScreen = pygame.image.load("assets/start_screen.png")
38. background = pygame.image.load("assets/background.png")
39. loseScreen = pygame.image.load("assets/lose_screen.png")
40. winScreen = pygame.image.load("assets/win_screen.png")
41. nextWave = pygame.image.load("assets/next_level.png")
42. finalWave = pygame.image.load("assets/final_level.png")
43.
44. # Ships
```

```
ship = ships.Player(windowWidth / 2, windowHeight, pygame, surface)
enemyShips = []

leftOverBullets = []

# Sound setup
pygame.mixer.init()

def launchWave():

  global lastSpawn, currentWave, currentLevel, gameOver, gameWon, nextLevelTS

  thisLevel = gameLevels.level[currentLevel]["structure"]

  if currentWave < len(thisLevel):

    thisWave = thisLevel[currentWave]

    for idx, enemyAtThisPosition in enumerate(thisWave):
      if enemyAtThisPosition is 1:
        enemyShips.append(ships.Enemy(((windowWidth / len(thisWave)) * idx), -60,
pygame, surface, 1))

  elif currentLevel + 1 < len(gameLevels.level) :
    currentLevel += 1
    currentWave = 0
    ship.shields = ship.maxShields
    nextLevelTS = timeTick + 5000
  else:
    gameWon = True

  lastSpawn = timeTick
  currentWave += 1

def updateGame():

  global mouseDown, gameOver, gameWon, leftOverBullets

  if mouseStates[0] is 1 and mouseDown is False:
    ship.fire()
    mouseDown = True
  elif mouseStates[0] is 0 and mouseDown is True:
    mouseDown = False

  ship.setPosition(mousePosition)

  enemiesToRemove = []

  for idx, enemy in enumerate(enemyShips):
```

```
93.
94.       if enemy.y < windowHeight:
95.         enemy.move()
96.         enemy.tryToFire()
97.         shipIsDestroyed = enemy.checkForHit(ship)
98.         enemyIsDestroyed = ship.checkForHit(enemy)
99.
100.        if enemyIsDestroyed is True:
101.          enemiesToRemove.append(idx)
102.
103.        if shipIsDestroyed is True:
104.          gameOver = True
105.          gameWon = False
106.          return
107.
108.      else:
109.        enemiesToRemove.append(idx)
110.
111.    oC = 0
112.
113.    for idx in enemiesToRemove:
114.      for remainingBullets in enemyShips[idx - oC].bullets:
115.        leftOverBullets.append(remainingBullets)
116.
117.      del enemyShips[idx - oC]
118.      oC += 1
119.
120.    oC = 0
121.
122.    for idx, aBullet in enumerate(leftOverBullets):
123.        aBullet.move()
124.        hitShip = aBullet.checkForHit(ship)
125.
126.        if hitShip is True or aBullet.y > windowHeight:
127.          del leftOverBullets[idx - oC]
128.          oC += 1
129.
130. def drawGame():
131.
132.    global leftOverBullets, nextLevelTS, timeTick, gameWon
133.
134.    surface.blit(background, (0, 0))
135.    ship.draw()
136.    ship.drawBullets()
137.
138.    for aBullet in leftOverBullets:
139.      aBullet.draw()
140.
141.    healthColor = [(62, 180, 76), (180, 62, 62)]
```

```
    whichColor = 0

    if(ship.health <= 1):
      whichColor = 1

    for enemy in enemyShips:
      enemy.draw()
      enemy.drawBullets()

    pygame.draw.rect(surface, healthColor[whichColor], (0, windowHeight - 5, (windowWidth / ship.maxHealth) * ship.health, 5))
    pygame.draw.rect(surface, (62, 145, 180), (0, windowHeight - 10, (windowWidth / ship.maxShields) * ship.shields, 5))

    if timeTick < nextLevelTS:
      if gameWon is True:
        surface.blit(finalWave, (250, 150))
      else:
        surface.blit(nextWave, (250, 150))

def restartGame():
  global gameOver, gameStart, currentLevel, currentWave, lastSpawn, nextLevelTS, leftOverBullets, gameWon, enemyShips, ship

  gameOver = False
  gameWon = False
  currentLevel = 0
  currentWave = 0
  lastSpawn = 0
  nextLevelTS = 0
  leftOverBullets = []
  enemyShips = []
  ship.health = ship.maxHealth
  ship.shields = ship.maxShields
  ship.bullets = []

def quitGame():
  pygame.quit()
  sys.exit()

# 'main' loop
while True:
  GAME_TIME.Clock().tick(30)
  timeTick = GAME_TIME.get_ticks()
  mousePosition = pygame.mouse.get_pos()
  mouseStates = pygame.mouse.get_pressed()

  if gameStarted is True and gameOver is False:
```

```
187.
188.     updateGame()
189.     drawGame()
190.
191.   elif gameStarted is False and gameOver is False:
192.     surface.blit(startScreen, (0, 0))
193.
194.     if mouseStates[0] is 1:
195.
196.       if mousePosition[0] > 445 and mousePosition[0] < 580 and
    mousePosition[1] > 450 and mousePosition[1] < 510:
197.         pygame.mouse.set_visible(False)
198.         gameStarted = True
199.
200.     elif mouseStates[0] is 0 and mouseDown is True:
201.       mouseDown = False
202.
203.   elif gameStarted is True and gameOver is True and gameWon is
    False:
204.         surface.blit(loseScreen, (0, 0))
205.         timeLasted = (gameFinishedTime - gameStartedTime) / 1000
206.
207.   if gameStarted is True and gameWon is True and len(enemyShips)
    is 0:
208.     surface.blit(winScreen, (0, 0))
209.
210.   # Handle user and system events
211.   for event in GAME_EVENTS.get():
212.
213.     if event.type == pygame.KEYDOWN:
214.
215.       if event.key == pygame.K_ESCAPE:
216.         quitGame()
217.
218.       if event.key == pygame.K_SPACE:
219.         if gameStarted is True and gameOver is True or
    gameStarted is True and gameWon is True:
220.           restartGame()
221.
222.   if timeTick - lastSpawn > gameLevels.level[currentLevel][
    "interval"] * 1000 and gameStarted is True and gameOver is
    False:
223.     launchWave()
224.
225.   if event.type == GAME_GLOBALS.QUIT:
226.     quitGame()
227.
228.   pygame.display.update()
```

projectiles.py

Download
magpi.cc/1jQlwov

```
01. class Bullet():
02.
03.   x = 0
04.   y = 0
05.   image = None
06.   pygame = None
07.   surface = None
08.   width = 0
09.   height = 0
10.   speed = 0.0
11.
12.   def loadImages(self):
13.     self.image = self.pygame.image.load(self.image)
14.
15.   def draw(self):
16.     self.surface.blit(self.image, (self.x, self.y))
17.
18.   def move(self):
19.     self.y += self.speed
20.
21.   def checkForHit(self, thingToCheckAgainst):
22.     if self.x > thingToCheckAgainst.x and self.x <
    thingToCheckAgainst.x + thingToCheckAgainst.width:
23.       if self.y > thingToCheckAgainst.y and self.y <
    thingToCheckAgainst.y + thingToCheckAgainst.height:
24.         thingToCheckAgainst.registerHit()
25.         return True
26.
27.     return False
28.
29.   def __init__(self, x, y, pygame, surface, speed,
30. image):
31.     self.x = x
32.     self.y = y
33.     self.pygame = pygame
34.     self.surface = surface
35.     self.image = image
36.     self.loadImages()
37.     self.speed = speed
38.
39.     dimensions = self.image.get_rect().size
40.     self.width = dimensions[0]
41.     self.height = dimensions[1]
42.
43.     self.x -= self.width / 2
```

LEARN TO LOVE THE COMMAND LINE

Get started today for **just £2.99 / $3.99**

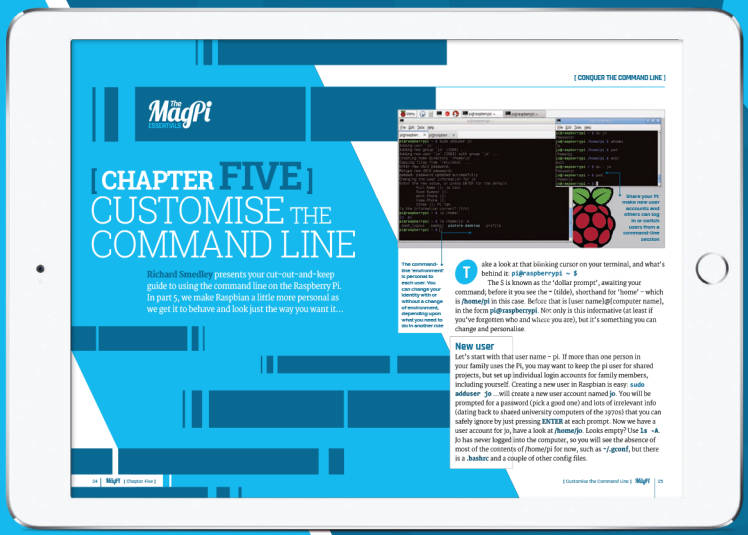
The MagPi ESSENTIALS

From the makers of the official Raspberry Pi magazine

Available on the App Store

Get it on Google play

Find it on The MagPi digital app


[CONQUER THE COMMAND LINE]
[CHAPTER FIVE]
CUSTOMISE THE COMMAND LINE
Richard Smedley presents your cut-out-and-keep guide to using the command line on the Raspberry Pi. In part 5, we make Raspbian a little more personal as we get it to behave and look just the way you want it...
Share your Pi: make new user accounts and others can log in or switch users from a command line session
The command-line 'environment' is different for each user. You can change your identity with or without a change of environment, depending upon what you need to do in another role
Take a look at that blinking cursor on your terminal, and what's behind it: pi@raspberrypi ~ $
The $ is known as the 'dollar prompt', awaiting your command; before it you see the ~ (tilde), shorthand for 'home' – which is /home/pi in this case. Before that is [user name]@[computer name], in the form pi@raspberrypi. Not only is this informative (at least if you've forgotten who and where you are), but it's something you can change and personalise.
New user
Let's start with that user name – pi. If more than one person in your family uses the Pi, you may want to keep the pi user for shared projects, but set up individual login accounts for family members, including yourself. Creating a new user in Raspbian is easy: sudo adduser jo ... will create a new user account named jo. You will be prompted for a password (pick a good one) and lots of irrelevant info (dating back to shared university computers of the 1970s) that you can safely ignore by just pressing ENTER at each prompt. Now we have a user account for jo, have a look at /home/jo. Looks empty? Use ls -a. Jo has never logged into the computer, so you will see the absence of most of the contents of /home/pi for now, such as ~/.gconf, but there is a .bashrc and a couple of other config files.
[Customise the Command Line] MagPi 25
24 MagPi [Chapter Five]

magpi.cc/Essentials-Bash

The MagPi ESSENTIALS

raspberrypi.org/magpi